

Modern Javascript

Lesson 1: **Introduction**

[About JavaScript](#)

[What is JavaScript?](#)

[JavaScript is Not Java](#)

[Using the CodeRunner® Editor](#)

[Saving and Retrieving Your Page](#)

[Previewing Your File](#)

[Looking Ahead...](#)

Lesson 2: **Your First JavaScript Script**

[Creating Your JavaScript Script](#)

[Where to Put Your Script](#)

[Linking to an External Script](#)

[The <script> Element](#)

[Create and Calling a Function](#)

[Built-In Functions vs. Creating Your Own Functions](#)

[Using Other People's Scripts](#)

[Script Libraries](#)

Lesson 3: **Variables**

[What is a Variable?](#)

[Values](#)

[Numbers](#)

[Strings](#)

[Booleans](#)

[Types](#)

[Naming Conventions for variables](#)

[Creating a Good Name](#)

Lesson 4: **Statements and Expressions**

[JavaScript Statements](#)

[The JavaScript Console](#)

[Arithmetic Operators](#)

[Increment and Decrement Operators](#)

[More Assignment Operators](#)

[Comparison Operators](#)

[Conditional Expressions and Statements](#)

[Statement nesting](#)

[The String Concatenation Operator](#)

[Comments](#)

Lesson 5: **Looping**

[The While Loop](#)

[The For Loop](#)

While or For?

Combining Looping Statements and Conditional Statements

BONUS: Using Loops to generate HTML

Lesson 6: **Arrays**

Creating Arrays and Accessing Their Values

Changing the Values in an Array

Looping with Arrays

Processing Arrays

Creating an Array from a String with Split

Searching for a String

Working with Numbers

Lesson 7: **The Document Object Model**

Behind the Scenes of a Web Page

The Document Object Model and document.getElementById()

Getting Access to Elements

Updating Elements

Getting Elements with document.getElementsByTagName()

Setting Up the window.onload and button.onclick Events

Get the Form Input Values and Update the Page

The Window Object

Lesson 8: **Functions and Events**

What is a Function?

Function Parameters and Arguments

Functions with Multiple Parameters

Multiple Functions

Handling Events

Functions as Event Handlers

Walking Through the Rest of the Code

Naming Functions

Lesson 9: **Scope**

Variable Levels

Global Scope and Local Scope

Local Scope

Shadowing

Scope of Function Parameters

Different Functions have Different Scope

Using Global and Local Variables

Lesson 10: **Objects**

Objects are Collections of Properties

Accessing Object Properties

Objects and Arrays

Storing Objects in an Array

[Arrays as Object Property Values](#)

[How are Objects and Arrays Similar and Different?](#)

[Object Constructors](#)

[Changing Object Properties](#)

[Built-In Objects](#)

[Element Objects](#)

[Using Objects to Collect Global Variables](#)

Lesson 11: **Methods**

[Objects and Methods](#)

[A Method is a Function in an Object](#)

[Methods with Parameters](#)

[This](#)

Lesson 12: **JavaScript and Forms**

[Create a Page with a Form](#)

[Add JavaScript to Process the Form](#)

[Set Up a Click Handler for the Button Input Control](#)

[Get the Value of a Text Input Control](#)

[A Slight Sidetrack: Logical Operators](#)

[Get the Value of a Numerical Input Control](#)

[Get the Value of a Select Input Control](#)

[Get the Value of a Text Area Control](#)

[Validate Form Input and Process the Data](#)

[Clearing Form Controls](#)

Lesson 13: **Form Collections**

[Using Form Collections](#)

[Using the Elements Collection to Access the Value of a Selected Radio Button](#)

[Submitting a Form with JavaScript](#)

Lesson 14: **Creating New Elements**

[Creating and Adding New Elements to the DOM](#)

[Adding More Elements](#)

[Inserting Elements](#)

Lesson 15: **Element Attributes and Style**

[Building the Photo Viewer Application](#)

[Add the CSS](#)

[Using `setAttribute\(\)` to Add a Class to an Element](#)

[Using `getAttribute\(\)` to Get the Value of an Element Attribute](#)

[Use `document.querySelectorAll\(\)` and `setAttribute\(\)` to Set the Class of Multiple Elements](#)

[Setting Style Directly Using JavaScript](#)

Lesson 16: **Navigating the DOM**

[Add a Click Handler to Multiple Links](#)

[Add an Image to the Page](#)

[Removing Elements from the Page](#)

[Getting a Child's Parent](#)

[Text Nodes](#)

[Avoiding Text Nodes Using getElementById\(\) or querySelectorAll\(\)](#)

Lesson 17: **[Final Project--The Amazing Box Generator!](#)**

[Amazing Boxes](#)

[The HTML for Amazing Boxes](#)

[The CSS for Amazing Boxes](#)

[Positioning an Absolutely Positioned Element](#)

[The Amazing Boxes Application Requirements](#)

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Introduction

Welcome to OST's introductory Javascript course! In this course, you will learn the basics of JavaScript programming and how to use it to suit your professional and creative goals.

Course Objectives

When you complete this course, you will be able to:

- develop the syntax and structure of JavaScript programs, including statements, expressions, variables, and operators.
- collect values using loops, arrays and objects.
- add and remove web page elements using the Document Object Model (DOM).
- validate and respond to user input using functions and events.
- create website menus with CSS and JavaScript.
- build a dynamic, interactive, front-end web application.

From beginning to end, you will learn by doing your own JavaScript based projects. These projects, as well as the final project, will add to your online portfolio and will contribute to your certificate completion.

About JavaScript

You're already familiar with HTML, but as a programming language, JavaScript is a more complex language than HTML. HTML is a markup language; a markup language is a set of markup tags. HTML uses markup tags to describe web pages. JavaScript is a scripting language designed to add interactivity to HTML pages.

It is an interpreted language (which means that scripts execute without preliminary compilation), and is usually embedded directly into HTML pages. If you're not familiar with programming languages, JavaScript might be challenging at first. There are many reserved words in this language. If you try to memorize each *word* you encounter in the course, you may feel overwhelmed. Instead, concentrate on concepts and the bigger picture:

- What is a JavaScript program?
- What is a loop, what is a conditional statement?
- When do I need an event handler?

We want you to gain knowledge of Javascript concepts and tools; if you get a handle on those, you'll be able to handle any weird and surprising JavaScript situations that may come your way. (Besides, you can always look up the meanings of the reserved words as they come up.) We used this same approach in our HTML class; each specific possible element was not covered, but students were presented with the right tools to be able to learn new elements and concepts as they arise.

What is JavaScript?

OK, so you've decided to learn JavaScript. So, what is JavaScript?

JavaScript, originally created by Netscape, is a language that's now supported by every major browser on the market (including Firefox, Chrome, Internet Explorer, Safari, and Opera) and many mobile browsers as well. The JavaScript language is completely devoted to creating **Web Applications**: applications that run in your browser as part of a web page. Because it can be used to create interactive web pages, communicate with web services, and create cookies, it is the most popular scripting language on the web. Browser makers have put a great deal of effort into improving the speed of JavaScript so it runs a lot faster than it did just a few years ago. That means you can now do more computation in a web page, enabling web applications like drawing programs and games.

Note

This course supports Mozilla Firefox, Google Chrome, Microsoft Internet Explorer, and Safari. If you're using a different browser, you may want to stop and start again using one of these.

JavaScript is an **Object-Based** language; it uses many of the concepts of Object-Oriented Programming, but it's not completely Object-Oriented.

JavaScript gives you (the programmer) more control over your web pages. JavaScript uses the document as

an environment where you can run applications.

JavaScript is an *interpreted* language. It is written and executed as is, and it does not need to be compiled or converted to another language before the browser can understand it.

JavaScript is Not Java

You may have heard of another programming language with a similar name: Java. Just in case, we should clear up any potential confusion.

Both JavaScript and Java are programming languages that are widely used in web development. Because the names are so similar, you might think that they're the same, or at least related. Actually, JavaScript is *not* Java; they're not even related!

Java is actually similar to the C++ programming language. Java is an Object-Oriented language derived from C++. You may also be familiar with Java Applets (programs written in Java that can only be run in a browser) on web pages. In fact, applets are only a limited portion of the capabilities of the Java language. Java was originally developed to be used as a programming language for word processors, calculators, car computers, watches, PDAs (Personal Digital Assistants), microwaves and such. The creators of Java wanted to invent a language to be used by multiple platforms. In the process of developing this language, they found that Java was ideal for the internet as well. The diversity of the Java language allowed it to work with the wide variety of platforms found on the Internet (UNIX, Windows, Macintosh, and more).

In addition, Java is a *compiled* (as opposed to *interpreted* like JavaScript) language. This means that the program is written in text, and converted to machine language (which you can't read) when it is executed.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add *looks like this*.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type *look like this*.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

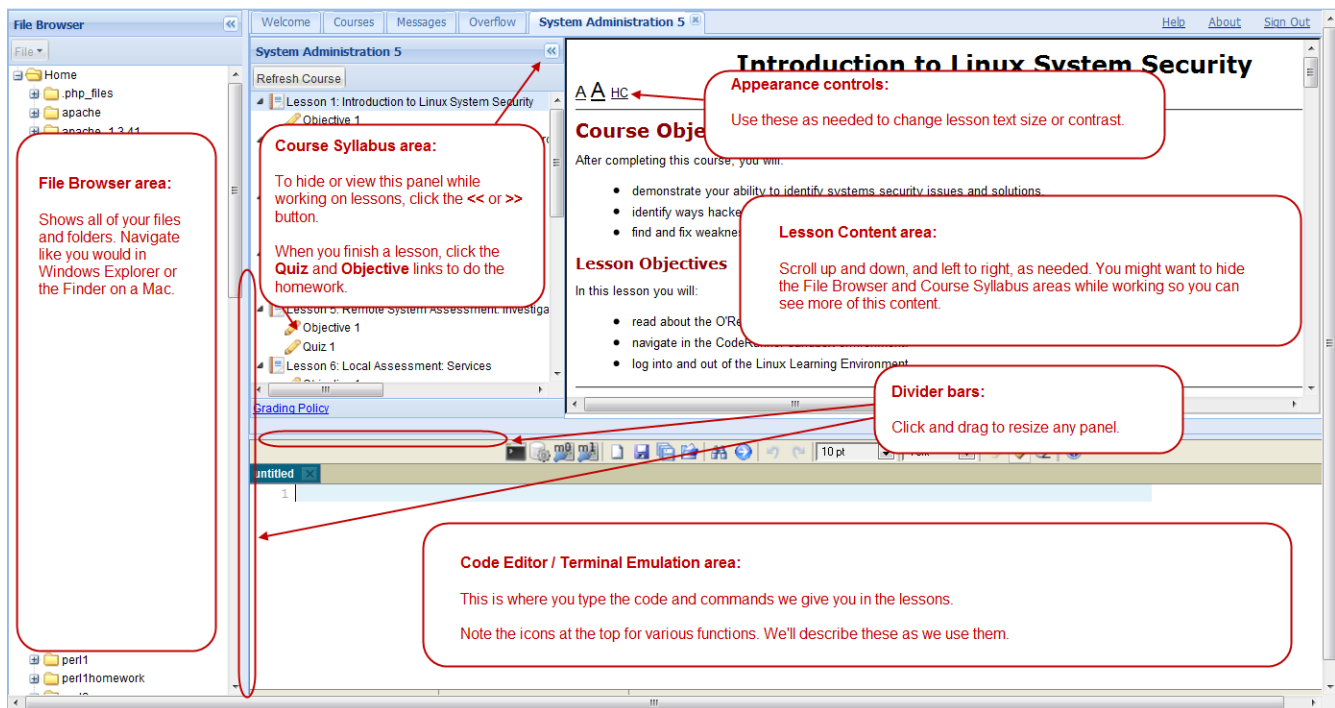
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

[File Management Demo](#)

[Code Editor Demo](#)

[Coursework Demo](#)

Using the CodeRunner® Editor

We're about to embark upon our first OST Learning Lab! You'll learn by trying the suggested lessons and experimenting on your own. In these Javascript labs, you'll script for the web and make your very own webpages!

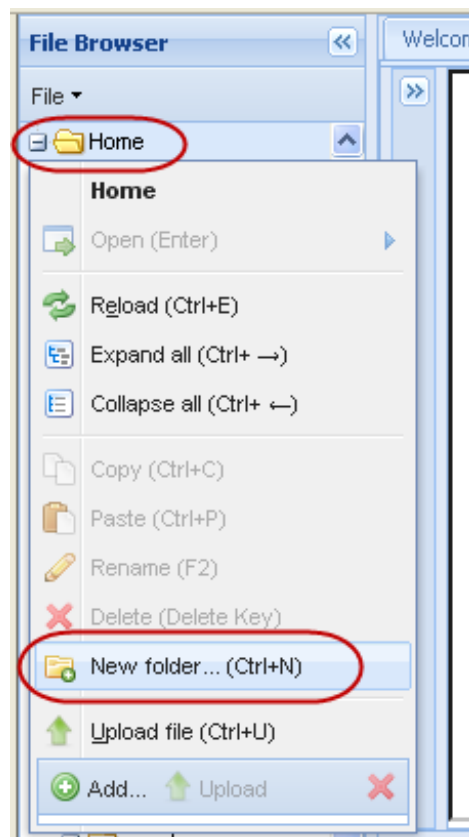
Let's take a closer look at the Editor below. In the bottom half of the window, you'll see the **CodeRunner® Editor**. Be sure that **HTML** is selected from the drop-down Syntax menu, and type the code as shown below:

CODE TO TYPE:

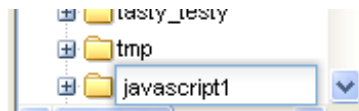
```
Hello, World!
```


Saving and Retrieving Your Page

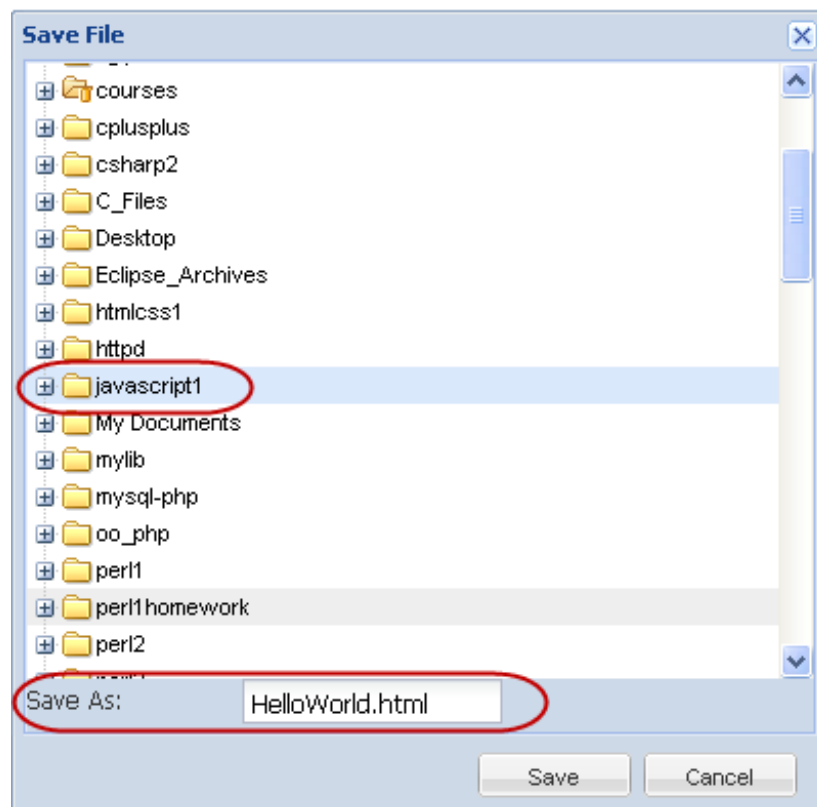
Now let's save your page on the OST server. First, we'll make a special folder to hold your work for this course. In the File Browser pane to the left of the lesson window, right-click the **Home** folder and select **New Folder**:



Enter the name **javascript 1**:




Now, to save your file, click on the  button. Select the new **/javascript 1** folder and then type the name **HelloWorld.html** in the Save As text box (you need to include the **.html** extension so the browser knows to treat your code as html):



You should also practice using the **Save as**  button to save another version of your file.

Good work! After successfully saving your file, anybody can go on the web, type the URL **`http://yourusername.oreillystudent.com/javascript1/HelloWorld.html`** in the location bar of their browser, and see the web page you've created.

To retrieve your page, simply click the **Load** icon  or double-click the file name in the File Browser window at the left.


Previewing Your File

After you save HelloWorld.html, click the **Preview** icon, which looks like this:



When you Preview a file, CodeRunner first checks to see whether this file has been saved before. If it has, the page will be saved with the same name. If not, the Save File window will open.

Note

Keep in mind that every time you Preview a file, your changes will be saved. Think about whether you want the previous code overwritten. If you don't, use File Save As  before you Preview.

Once the file has been saved, another browser window or tab will open and show you your first web page.

Looking Ahead...

Now that you know a little about it, you're ready to learn some JavaScript! By the time you've finished this course, you'll be able to apply the powerful abilities of JavaScript to your documents. So, what kind of stuff will you be able to do? Well, take a look at this example. With a little hard work, you'll be able to create something like this and a whole lot more:

OST Tic Tac Toe

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Your First JavaScript Script

Lesson Objectives

When you complete this course, you will be able to:

- demonstrate your ability to create a script.
- place your script in the appropriate location.
- link to an external script.
- use the script element.
- create and call a function.
- use built-in functions.
- use scripts created by others.
- access script libraries.

Creating Your JavaScript Script


We've talked about JavaScript long enough! Let's dive right in and write your first script. Set the Syntax to **HTML** and type the code into the editor, as shown:

CODE TO TYPE:

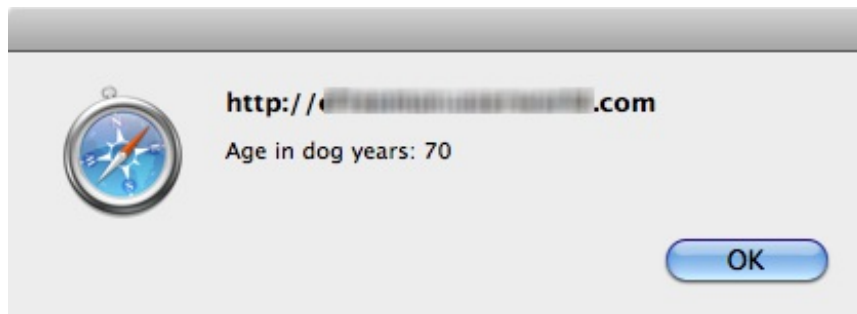
```
<!doctype html>
<html lang="en">
<head>
  <title> My First JavaScript </title>
  <meta charset="utf-8">

  <script>
    var age = 10;
    var ageInDogYears = age * 7;
    alert("Age in dog years: " + ageInDogYears);
  </script>

</head>
<body>
</body>
</html>
```

Save the file in your **/javascript 1** folder as **dogyears.html**, and click **Preview** .

You'll see an alert dialog popup; it looks like this:



Where to Put Your Script

While the HTML probably looks familiar to you, we also introduced a new element—the **<script>** element. This element tells the browser to expect JavaScript. In this case, we placed the **<script>** element in the *head* of our HTML document. That's generally a good place for a short script that will be used only by the page on which it's located.

You can also place the script in the *body* of your HTML if you like. Modify your code as shown:

CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> My First JavaScript </title>
  <meta charset="utf-8">

  <script>
var age = 10;
var ageInDogYears = age * 7;
alert("Age in dog years: " + ageInDogYears);
  </script>

</head>
<body>

  <script>
    var age = 10;
    var ageInDogYears = age * 7;
    alert("Age in dog years: " + ageInDogYears);
  </script>

</body>
</html>
```

Preview  Save and preview it.

The browser runs your JavaScript code as soon as it sees it, while it's parsing your HTML document. So, if you place your JavaScript in the `<head>` of your document, it will run earlier than if you place it in the `<body>`. In JavaScript, like in HTML, the browser evaluates and runs your code top-down, so JavaScript statements at the top run before JavaScript statements at the bottom.

As you'll see a bit later, ordering is important!

Linking to an External Script

If you're going to use your JavaScript in multiple HTML files, or you just want to keep your JavaScript and HTML separate, you can link to a JavaScript file. It's similar to how you link to CSS files, except that instead of using the `<link>` tag, you use the `<script>` tag. First, copy and paste all the JavaScript (between the `<script>` tags) from `dogyears.html` into a new file in your `/javascript 1` folder; name that file **dogyears.js**. Then, remove that JavaScript from `dogyears.html` and change the `<script>` element so that it links to the new file. Modify your code as shown:

CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> My First JavaScript </title>
  <meta charset="utf-8">

  <script src="dogyears.js">
</script>

</head>
<body>
  <script>
    var age = 10;
    var ageInDogYears = age * 7;
    alert("Age in dog years: " + ageInDogYears);
  </script>

</body>
</html>
```

We moved the `<script>` element back to the `<head>` of the document. Even though we don't have any JavaScript between the opening and closing script tags, we still include both. This is important! If you don't include both the opening and closing script tags, you may get weird behavior in your browser.

Click  on `dogyears.html`. You should see the same dialog as before.

Linking to an external JavaScript file from the `<head>` of the document is the most common way to include JavaScript in a web page. If you "view source" code of web pages, you'll usually see the JavaScript included like that. We'll be using the same method frequently, as well as placing our JavaScript in the `<head>` element, throughout the rest of the course.

Try it right now! Go to the [O'Reilly School website](#) and view the page source. You'll see a combination of links to JavaScript and JavaScript embedded in the document itself.

The `<script>` Element

The `<script>` element is pretty straightforward and you now know everything you need to know about `<script>` for this course. But, when you check out how other pages link to JavaScript files, you might notice that some links include a **type** attribute.

The **type** attribute used to be required because browsers used to support different scripting languages; for instance, for a long time, IE supported JScript, while Netscape supported JavaScript. Now that browsers have standardized a bit more, JavaScript is the default scripting language for all the major browsers, so the type attribute is optional. Still, if you'd like to use it (maybe you know that some of your users still use old browsers), you can—it's still supported! Edit `dogyears.html` as shown:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> My First JavaScript </title>
  <meta charset="utf-8">
  <script src="dogyears.js" type="text/javascript">
</script>
</head>
<body>
</body>
</html>
```

Create and Calling a Function

When you're writing JavaScript, most of the time you'll put your code in *functions*. You'll also use many of the built-in

functions available in JavaScript. We'll cover functions in greater detail later in the course, for now, think of a function as a way to package a chunk of code so that it's more convenient to access and reuse later.

Just to get warmed up using functions, let's change the script you've already written in **dogyears.js**, and put that code inside a function. Modify the code as shown:

CODE TO TYPE:

```
function dogYears() {  
    var age = 10;  
    var ageInDogYears = age * 7;  
    alert("Age in dog years: " + ageInDogYears);  
}
```


We've put all the code that was in your file into a function named **dogYears**. The **function** keyword indicates that we're defining a function named **dogYears**. All the code in the function goes between an opening and closing brackets **{}**. By convention, we indent the code inside the function, so it's easier to see that the code is part of the function and not the code around it, and also to keep the brackets matching.

Save your **dogyears.js** file, and click  on **dogyears.html** to reload the page. Did you see the alert?

No! You did *not* see the alert because now all the JavaScript code is in a function, and the only way to get the function's code to run is to *call* it. To do that, add one more line of code to **dogyears.js**. Modify your code as shown:

CODE TO TYPE:

```
function dogYears() {  
    var age = 10;  
    var ageInDogYears = age * 7;  
    alert("Age in dog years: " + ageInDogYears);  
}  
dogYears();
```


Save your JavaScript file, and  on **dogyears.html** again. Do you see the alert now?

Now that you've packaged the code into a function, it'll be convenient to use that code whenever you want—behold the power of functions! You could type **dogYears()**; as many times as you want in your JavaScript file and you'd get the equivalent number of alerts. You can reuse the code in the **dogYears()** function without having to type it repeatedly.

Try adding a few more calls to the **dogYears()** function as shown:

CODE TO TYPE:

```
function dogYears() {  
    var age = 10;  
    var ageInDogYears = age * 7;  
    alert("Age in dog years: " + ageInDogYears);  
}  
dogYears();  
dogYears();  
dogYears();  
dogYears();
```

 **dogyears.html** again How many alerts do you see now? It might seem like overkill to get that many alerts, but you can see how nice it is to have that code packaged up into a function.

Built-In Functions vs. Creating Your Own Functions

Most of what you do in JavaScript is writing and using functions. For example, **alert** is a built-in function that takes a string (a series of characters between quotation marks) and displays that string in a dialog box. Let's try another built-in function, **prompt**. Type this code into your editor as shown:

CODE TO TYPE:

```
function dogYears() {  
    var age = 10;  
    var age = prompt("Enter your dog's age: " );  
    var ageInDogYears = age * 7;  
    alert("Age in dog years: " + ageInDogYears);  
}  
dogYears();  
dogYears();  
dogYears();  
dogYears();
```

Note

If you are using IE, you may need to change a browser setting for this script to work properly. Select **Tools | Internet Options | Security** and check that "Allow websites to prompt for information using scripted windows" is enabled.

Preview

dogyears.html to see how **prompt** works. It prompts you to enter your dog's age. So instead of just setting the dog's age to 10 every time we call the function dogYears, we ask the user to enter their dog's age, storing that value in a variable named age, and then displaying that age multiplied by 7.

Don't worry about all the details of variables and functions yet; we're just getting your feet wet with JavaScript and getting a basic script working for now. We'll come back to all these details shortly, I promise!

Using Other People's Scripts

Just about every web page on the internet uses JavaScript. If you find a page that contains JavaScript you like, it may be convenient to copy and paste the JavaScript from the web page into yours and use it. That's absolutely fine when you're learning JavaScript and you want to try things out. But it's definitely *not* fine for you to use other people's scripts in a live web page without getting their permission first!

If you find a script you like, and you want to use it, make sure you ask permission from the person upon whose web page you found it. Typically, people are open to sharing their scripts, as long as you give them attribution for them. Using scripts without permission is copyright infringement, so stay on the safe (and honorable) side, and ask permission before you use other people's scripts.

Script Libraries

There are lots of places you can get *script libraries* now as well. These are scripts that are designed to be downloaded, or linked to, and used by other people. One of the most popular script libraries on the internet is **jQuery**, a library designed to make many common tasks you do with JavaScript (to interact with a web page, or create UI effects, for instance) easier. You can use this library without asking anyone's permission, as long as you leave all the information in the JavaScript files you download intact so others know you're using the jQuery library.

Perform a *view source* on a few of your favorite web pages and see if you can find the JavaScript. You might find that some of them link to well-known libraries, like jQuery, or have their own code in the page or linked to from the page, or both!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Variables

Lesson Objectives

When you complete this course, you will be able to:

- declare variables.
 - enter values into variables.
 - change variable types.
 - use standard naming conventions for your variables.
-

What is a Variable?

A variable is a container for a value. You've actually experimented with variables a bit already in this course. It looked something like this:

```
var age = 10;
```

In this example **var** is a *keyword*, a special word that tells JavaScript you're **declaring** a variable. The variable is **age** and the value you're putting into the **age** container is the number **10**. When you write **var age = 10;** you're actually doing *two* things: you're *declaring* the variable (with **var**) and you're *initializing* it (by setting its value to 10). You can also just declare a variable without initializing it, like this:

```
var age;
```

In this example, the variable's value is undefined until you give it a value, so it's more likely that you'll declare and initialize a variable at the same time.

You only need the **var** keyword when you *declare* a variable. "Declaring a variable" means you're telling JavaScript about it for the first time, and usually giving that variable a value.

Note

You may have seen scripts that don't use the **var** keyword to declare variables (the **var** keyword is omitted). JavaScript is pretty forgiving and will let you do this, but it's not a good idea because of certain assumptions that are made about the variable. We'll come back to this later; for now, just get in the habit of using the **var** keyword whenever you declare a variable!

Values

So, what kinds of values can you put into variables?

Numbers

You can put *numbers* into variables. Try this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title>Storing Numbers</title>
  <meta charset="utf-8">
  <script>
    var x = 3;
    alert("x = " + x);

    var y = 3.0;
    alert("y = " + y);

    var z = 3.1;
    alert("z = " + z);

    var big = 3.0E12;
    alert("big = " + big);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **var_number.html** and click . Make note of the values you see. Are they what you expected?

In JavaScript, there are two kinds of numbers: integers and floating point. You probably recall from your school days that integers are positive and negative whole numbers (...-2, -1, 0, 1, 2, 3...). To declare a variable **x** with the value **3**, you would type:

OBSERVE:

```
var x = 3;
```

To declare a variable **y** with the value **-3**, you would type:

OBSERVE:

```
var y = -3;
```

Floating point numbers are real numbers (that is, a number that can contain a fractional part), like **1.1**, **-3.145**, and **36034.55**. To declare these numbers, you would type, for example:

OBSERVE:

```
var x = 1.1;
var y = -3.145;
var z = 36034.55;
```

Floating point numbers don't always have to contain a decimal point. In fact, in JavaScript, most of the time you don't have to don't need to be concerned with whether a number has a decimal point; you can just use it as a number. There are some exceptions to that rule though; we'll go over them later.

So, suppose you want to represent a number like **3,000,000,000,000** (that's 3 trillion). You could write out the whole long thing like this:

OBSERVE:

```
var big = 3000000000000;
```

Or, you could use *scientific notation*, and write it like this:

OBSERVE:

```
var big = 3.0E12;
```

The **E12** tells JavaScript to multiply 3.0 by 1 followed by 12 zeros (or 1 trillion). You probably won't use this notation too often, but once in a while it can come in handy!

Note Make sure you leave out the commas when you're writing a number value in JavaScript!

Strings

Another type of value you can put into JavaScript variables is a *string*. A string is a sequence of characters with quotation marks around it, like this: **"Fido"**, or this:

OBSERVE:


```
var dogLikes = "Fido likes to go for long walks";
```

A string may contain many characters, including spaces, numbers, and even other quotes.

Suppose you want to write a string containing another string in JavaScript. For instance, you want to include the string: "And the President said, "I cannot tell a lie." "; Including quotation marks in your string presents a challenge. (We are not challenged by punctuation, by the way. We're just using two sets of quotation marks for the sake of our example, sacrificing sound grammar for technology. Sometimes it has to be this way.) How could you do it? Try experimenting with strings and quotation marks. Try typing a string that contains double quotation marks:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> A famous quote </title>
  <meta charset="utf-8">
  <script>
    var quote = "And the President said, "I cannot tell a lie." ";
    alert(quote);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **var_string.html**, and click **Preview** . Do you see the alert? Probably not, because the code causes an error.

When Javascript sees the first quotation mark before the word **And**, it recognizes it as the beginning of a string. When JavaScript gets to the quotation mark in front of the word **I**, it thinks it's the *end* of the string. The word **I**, and all the words following it, aren't valid JavaScript, so those words cause an error.

Note When you display the string value in a variable, you won't see double quotation marks; they are only present to tell JavaScript where the string begins and ends.

To place quotations inside a string, you need to *escape* them, by preceding them with a special character, the backslash (****):

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> A famous quote </title>
  <meta charset="utf-8">
  <script>
    var quote = "And the President said \"I cannot tell a lie.\" ";
    alert(quote);
  </script>
</head>
<body>
</body>
</html>
```


Save it and click . Now your alert works fine.

The backslash tells JavaScript to "escape" from its normal interpretive process and treat the double quotation marks that follow as a punctuation mark rather than as a string delimiter.

You can also use an apostrophe within a string, like this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> A famous quote </title>
  <meta charset="utf-8">
  <script>
    var quote = "It's no exaggeration to say that the undecideds could go one way or another.";
    alert(quote);
  </script>
</head>
<body>
</body>
</html>
```


Save it and click . You'll see the string in the alert.

We're using an apostrophe in the word **It's**. We don't have to use a backslash in front of it because it won't cause any problems. JavaScript knows that the string doesn't end until it sees another set of quotation marks.

One last bit of information to keep in mind regarding strings: if you need to type a really long string, you should write it all on one big long line, without any carriage returns. Give it a try. Type the code below as shown:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> A famous quote </title>
  <meta charset="utf-8">
  <script>
    var quote = "Thomas Jefferson once said, \"We should never judge a president by his age, only by his works.\"";
    alert(quote);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click .

JavaScript expects a string to start and end on the same line; if you split your string into two separate lines like this, JavaScript will get confused! You'll see shortly how you can split up strings over multiple lines. For now, just keep typing your strings in one continuous line.

Booleans

The other basic value you can put into a JavaScript variable is a *Boolean*. A Boolean value is **true** or **false**. You can write it like this:

OBSERVE:

```
var isBig = true;
var isOld = false;
```

We don't include quotation marks around the values **true** and **false**.

You'll learn about Boolean values in greater detail later, but for now, try this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Booleans </title>
  <meta charset="utf-8">
  <script>
    var age = 14;
    var isYoung = age < 21;
    alert("isYoung = " + isYoung);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **var_boolean.html**. Notice that instead of setting the variable **isYoung** directly to a Boolean **true** or **false** value, we set it to the result of an *expression*. Can you guess what this code does? Try it out.

What happens if you change the value of **age** to 22, or if you change the number you're comparing **age** to, from 21 to 13? Do some experimenting with Booleans!


Types

Even though we talked about the types of the values (number, string, or Boolean) we put into our variables, there is nothing in the variable declaration that indicates which type the variable should be. An unassigned variable like **var x** could contain any of those types.

In other languages you must indicate the type of the variable in order to tell the computer which kind of value should be in that variable; this is not required in JavaScript. This characteristic is known as *dynamic typing*; it means that JavaScript is pretty good at guessing the type from the value. It also means that if you really wanted to, you could *change* the type of a variable halfway through a program, like this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Types </title>
  <meta charset="utf-8">
  <script>
    var x = 3;
    alert(x);
    x = "dog";
    alert(x);
  </script>
</head>
<body>
</body>
</html>
```

We initialize the variable `x` to the number 3, and then change its value to a string, "dog." Save it in your `/javascript1` folder as `var_boolean.html` and click [Preview](#) . Does it work?

This is perfectly legal in JavaScript, but it's not a particularly good idea. Why? Well, suppose you have a big program that first initializes `x = 9`;, and then you write code that expects `x` to be a number, like, `x = x + 10`;, and *then*, you set `x = "W"`;, and *forget* your earlier `x = x + 10`; code—if you happen to loop back to that code again and try to add 10 to "W," JavaScript will not be pleased!

Sticking with one type for a variable is a good idea and will help reduce the number of bugs you need to solve! It will also make it a lot easier to read your code later when you're trying to remember how or why you did something.

Which leads us to our next topic...

Naming Conventions for variables

We've talked a lot about variables and the kinds of values you can store in them, but we haven't talked much yet about the kinds of names you can use for your variables. We've used names like `x` and `y`, as well as `quote` and `isBig`. All of these are either letters or strings, which brings us to the first of three rules you need to know to create good variable names:


Rule 1: Begin your variable names with a letter.

All variable names must begin with a letter. You're also allowed to use `"_"` and `"$"` to begin a variable name, but because those characters are often used by variables in JavaScript libraries, it's best not to use them to begin a variable name unless you really know what you're doing. We'll just stick with letters. Try these:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Variable Names </title>
  <meta charset="utf-8">
  <script>
    var myName = "Scott";
    var isRaining = true;
    var _libVar = 0;
    var $importantVariable = 99;


    alert(myName);
    alert(isRaining);
    alert(_libVar);
    alert($importantVariable);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **var_names.html** and click . You'll see all the alerts with their correct values. What happens if you type these variable names instead? Try this code:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Variable Names </title>
  <meta charset="utf-8">
  <script>
    var 5year = 5;
    var %interest = 9.89;
    var ~test = false;

    alert(5year);
    alert(%interest);
    alert(~test);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . Do you see any alerts? You probably don't, because these are *not* valid variable names!

Rule 2: After the first character, you can use letters, numbers, underscores, and dollar signs.

Once you've got the first character of your variable set, then you can use letters, numbers, "_," and "\$" as much as you like. Variable names can be as long as you like, and while you don't want to go overboard with super long names, it's a good idea to have *reasonably* long, descriptive names for variables—descriptive names make it easier to understand your code. Try these valid names:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Variable Names </title>
  <meta charset="utf-8">
  <script>
    var amount$ = 9.99;
    var tax35 = 35;
    var keep_real = true;

    alert(amount$);
    alert(tax35);
    alert(keep_real);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . You'll see alerts.

As long as you stick with letters, numbers, "_," and "\$" for your variable names, you're good to go. Try using these variable names:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Variable Names </title>
  <meta charset="utf-8">
  <script>
    var twoand1/2men = "crazy";
    var last-name = "Gray";
    var two parts = true;
  </script>
</head>
<body>
</body>
</html>
```

Save it and click .

All of these variables have characters that aren't allowed in variable names because those characters have other meanings.

Note There's a common convention of using underscores (`_`) to represent spaces in variable names, so `"two_parts"` might be a better choice for that last one.


Also, variable names are *case sensitive*, which means that a variable named `tax35` is a *different variable* from one named `TAX35`.

Rule 3: Avoid using JavaScript's reserved keywords and built-in function names.

We've already encountered one keyword in this lesson: `var`. It would be mighty confusing to both you and JavaScript if you wrote `var var!` Try it:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Variable Names </title>
  <meta charset="utf-8">
  <script>
    var var;
    alert(var);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . You don't see anything, right? JavaScript will definitely complain if you do this, so stay away from JavaScript's keywords. You'll get familiar with them fairly quickly and most of them make sense once you get to know the language. You can find a [comprehensive list](#) of all the reserved words online; here are a few:

- `boolean`
- `break`
- `catch`
- `char`
- `class`
- `continue`
- `default`
- `delete`

- do
- double
- else
- false
- final
- float
- for
- function
- goto
- if
- interface
- long
- namespace
- new
- null
- return
- super
- this
- throw
- true
- try
- var
- while
- with

We'll encounter several of these reserved words throughout the course.

Creating a Good Name

As we've already mentioned, it's a great idea to use *meaningful* names so when you go back to read your code later, you'll know what you meant!

Another good tip for making variable names, is to use *camel case*. Camel case means that the first letter of each word (except the first one) is capitalized within the variable name, so the capital letters look like the humps of a camel. Like camelCase (a one-hump camel) or myFavoriteBook (a two-hump camel). By convention, variable names usually begin with a lower case letter except in some special circumstances.

Finally, remember to start your variables with a letter, rather than "_" or "\$" unless you have a very good reason (you'll know when you do). This helps you to avoid variable name clashes with JavaScript libraries.

You've got the low-down on variables now, so go make some! Try your hand at creating good names, and use **alert** to test them out. Have fun! In the next lesson you'll learn all about how to *do* more with variables, creating expressions and statements. See you there!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Statements and Expressions

Lesson Objectives

When you complete this course, you will be able to:

- use the JavaScript console.
- incorporate basic JavaScript statements into your web pages.
- use various JavaScript operators including arithmetic, increment, decrement, assignment, comparison and string concatenation operators.
- use conditional expressions and statements.
- nest your statements.
- comment your JavaScript code appropriately.

JavaScript Statements

Each line of JavaScript code that you write is called a *statement*. For example, `x = 3 + 2;` is a statement, as is `alert(x);`. Throughout the course, you'll learn about other kinds of statements like `if` statements, `while` loops, and so on. A JavaScript program is a sequence of statements.

Take a look at this one:

OBSERVE:

```
x = 3 + 2;
```

It's made up of four parts: `x`, `=`, `3 + 2`, and the `;` at the end. All JavaScript statements end in a **semicolon**. The part of a statement that has a value, like `3 + 2`, is called an *expression*, and its *result*, in this case the value 5, is *assigned* to the variable `x`.

Remember our first JavaScript program? It contained these two statements:

OBSERVE:

```
var age = 10;  
var ageInDogYears = age * 7;
```

Notice that the right side of each statement is an expression. In the first statement, the result of the expression is just 10. In the second statement, JavaScript determines the result of the expression `age * 7` by taking the value of the variable `age` and multiplying it by 7. So the value of the variable `ageInDogYears` is 70 after JavaScript runs that statement.

These kinds of statements, with an `=` sign, are called *assignment* statements; there's a variable on the left side of the `=`, and a value or an expression that gets evaluated on the right side of the `=`, and the resulting value is *assigned* to the variable.

`var x = 3;` is also an assignment statement; in this case you're declaring the variable `x` and initializing, or *assigning*, the value 3 to it.

The JavaScript Console


Before you learn about any more JavaScript statements and expressions, it's time you learned how to use the JavaScript console. You can try all these statements and expressions by typing them into an HTML file and using alerts, but sometimes it's easier just to type them right into a console window. The other good reason to learn how to use the console window is that you can more easily find errors in your code! If you make an error in your JavaScript code, and load it into a browser window, unless you're using the developer console, chances are you'll never know you've got an error except for the fact that your page isn't doing what you expect. Using the console, you can get information about the error, and even what line number the problem is on.

One of the most common ways you'll use the console is with the `console.log` statement. Until now, we've been using `alert` to test our code and see values of variables and expressions, but `console.log` is a better way to do this.

Let's create a quick program to see how it works.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Console Test </title>
  <meta charset="utf-8">
  <script>
    var x = 3;
    console.log("x is: " + x);
  </script>
</head>
<body>
  Testing the console!
</body>
</html>
```

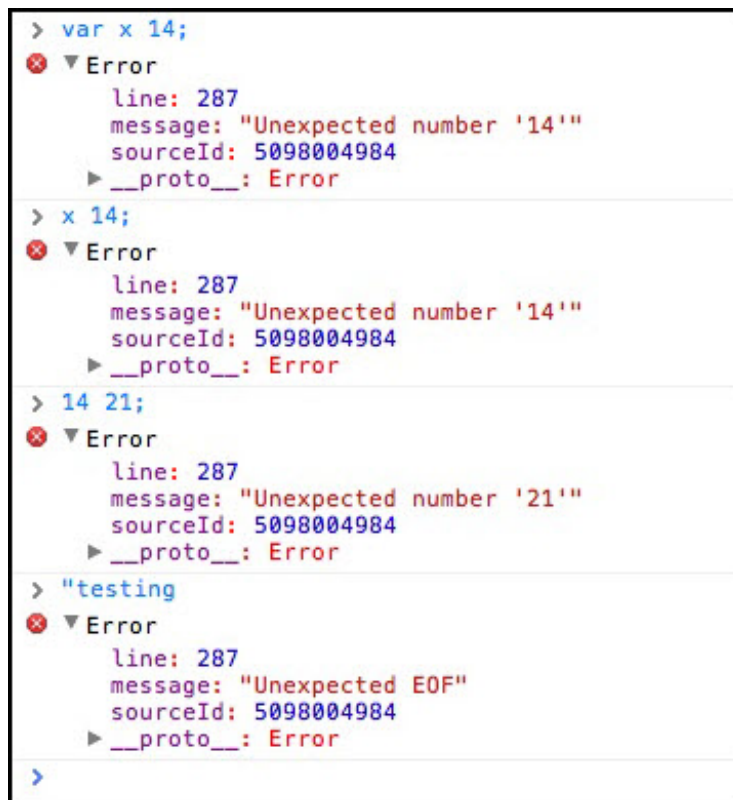
Save it in your **/javascript1** folder as **consoletest.html** and click **Preview** . It displays the string "x is 3" in your console, showing you the value of the variable. This is a great debugging tool! Before we write any code that uses **console.log**, check out the guide and video on how to access the JavaScript console using the Developer Tools in the browser of your choice.

The developer tools are a little different in every browser, and are changing frequently as new browser versions come out, so you might have to be a little industrious and do some experimenting on your own. We've created a very basic [guide](#) to get you started, and also a short video explaining how to access the developer tools, for [Safari](#), [Chrome](#), and [Firefox](#).

...and for [Microsoft Internet Explorer 9](#).

Once you've got the developer console up and running in the browser of your choice, try typing in some statements and expressions and see what happens. Also, try making some errors, so you get used to the kinds of error messages you might see in your JavaScript programs.

Here are some examples of typing errors into the console:



```
> var x 14;
✖ Error
  line: 287
  message: "Unexpected number '14'"
  sourceId: 5098004984
  __proto__: Error

> x 14;
✖ Error
  line: 287
  message: "Unexpected number '14'"
  sourceId: 5098004984
  __proto__: Error

> 14 21;
✖ Error
  line: 287
  message: "Unexpected number '21'"
  sourceId: 5098004984
  __proto__: Error

> "testing
✖ Error
  line: 287
  message: "Unexpected EOF"
  sourceId: 5098004984
  __proto__: Error

>
```

If "undefined" shows up in the console, don't worry about it. It's just JavaScript showing you the result of typing in a statement like **var x = 3;**. The result of the *statement* is undefined, even though the result of the expression is 3, and the

variable `x` has the value 3. You can always test the value of `x` by typing `x` at the console prompt, and you should see 3 in this case. Here's how that would look in the console:

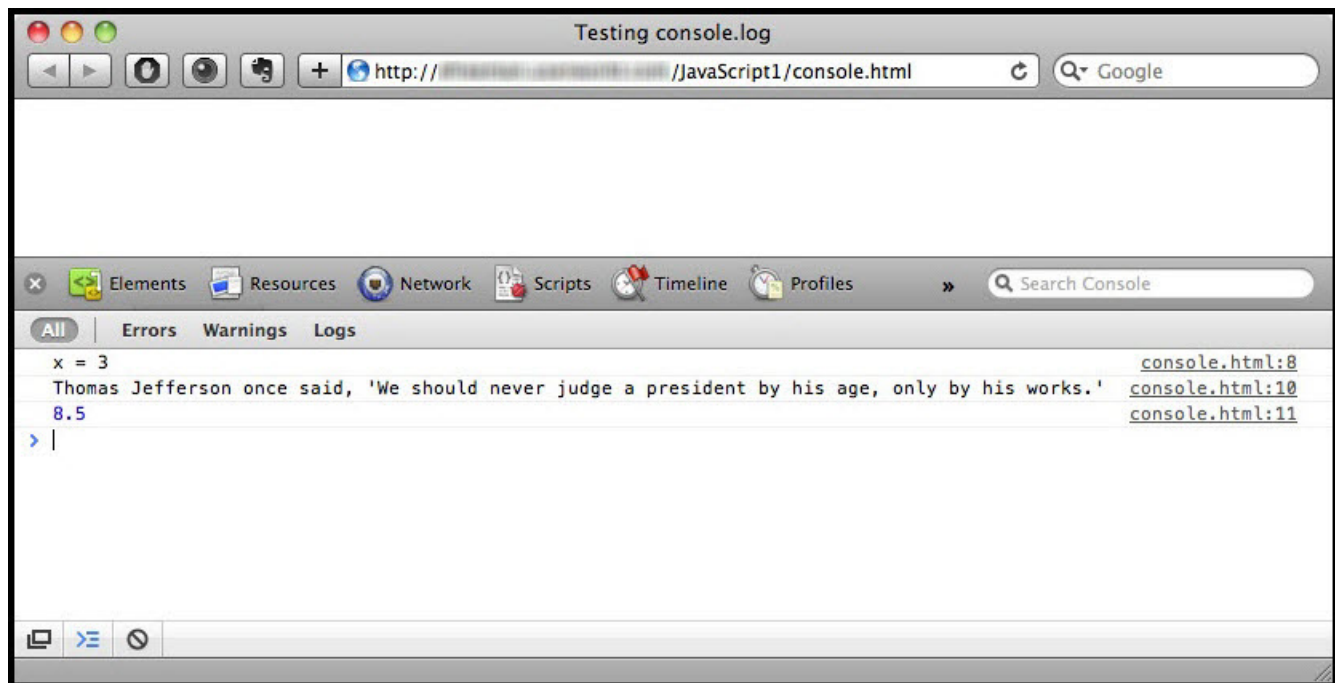
```
> var x = 3;
undefined
> x
3
>
```

Let's modify our program and use **console.log** to display the values of some expressions and variables in the console.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Console Test </title>
  <meta charset="utf-8">
  <script>
    var x = 3;
    console.log("x is: " + x);
    var quote = "Thomas Jefferson once said, 'We should never judge a president by his
age, only by his works.'";
    console.log(quote);
    console.log(6.5 + 2);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click **Preview** .



Note

You're welcome to continue using **alert()** instead of **console.log()** if you prefer, or if you haven't been able to get the console working yet.

Arithmetic Operators

You're probably familiar with many of the *arithmetic operators* available in JavaScript, like addition (+), subtraction (-), multiplication (*) and division (/). Let's take a look at some of these operators in action:


CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arithmetic Operators </title>
  <meta charset="utf-8">
  <script>
    var length = 100;
    var sideOfSquare = length / 4;
    console.log(sideOfSquare);

    var discountPercent = 3;
    var retailCost = 49.95;
    var cost = retailCost - (retailCost * discountPercent/100);
    console.log(cost);

    var radius = 3;
    var circumference = 2 * Math.PI * radius;
    console.log(circumference);

  </script>
</head>
<body>
</body>
</html>
```

Save it in your `/javascript1` folder as `arithmetic.html` and click . You should see three values in the console:

```
25
48.4515
18.84955592153876
> |
```

Let's look at the script in detail:

OBSERVE:

```
var length = 100;
var sideOfSquare = length / 4;
console.log(sideOfSquare);

var discountPercent = 3;
var retailCost = 49.95;
var cost = retailCost - (retailCost * discountPercent/100);
console.log(cost);

var radius = 3;
var circumference = 2 * Math.PI * radius;
console.log(circumference);
```

The first value (**25**) is the result of the variable **length** value (100) divided by 4.

The second is more complex because we combined multiple operators in one expression. Do you remember rules of *precedence* from algebra class? It's a good idea in JavaScript to use parentheses when combining operators so there's no ambiguity about which operators to evaluate first! In the case of computing the value of the variable **cost**, we tell JavaScript to first evaluate the expression **retailCost** (**49.95**) * **discountPercent** (**3**)/**100**, and then subtract the result of that from the value of **retailCost** (again, **49.95**: $49.95 - (49.95 * 3/100) = 48.4515$).

Finally, in the last example, we multiply a built-in JavaScript value, **Math.PI**, by a **radius** value to compute the

circumference of a circle. **Math** is available in all browsers and has some common constants that may come in handy if you do a lot of mathematical computation in your programs.

Try some other arithmetic expressions, including the **modulus** operator (%). This operator is used to find the remainder of a division. Since any even number divided by 2 gives no remainder, one way you can use this operator is to determine whether a number is odd or even.

OBSERVE:

```
var age = 49;
var oddOrEven = age % 2;
console.log(oddOrEven);
```

Because the variable **age** is odd, the variable **oddOrEven** will contain the value 1 (the remainder of dividing 49/2).

Increment and Decrement Operators

Two other operators that you'll find useful are the increment (**++**) and the decrement (**--**). Try this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Increment and Decrement Operators </title>
  <meta charset="utf-8">
  <script>
    var length = 100;
    length++;
    console.log("Length after ++ is: " + length);
    length--;
    console.log("Length after -- is: " + length);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **increment.html** and click [Preview](#). Do the values in the console match what you expect? You should see the values 101 and 100 as the results of the two console.log statements.


++ just adds 1 to the value of a numeric variable, so **x++**; is equivalent to **x = x + 1**; and says, take the current value of x, add 1 to it, and store the new value back in x. Likewise, **--** subtracts 1 from the value of a variable. You'll see these in use a lot with loops, which we'll get to in the next lesson.

More Assignment Operators

You've already seen one assignment operator, the equals sign (=), which is used in assignment statements to assign a value to a variable. There are quite a few more assignment operators that all combine assignment with an arithmetic operator, like +. For instance, the **+=** combines = and + into one operator. Modify your file as shown:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Assignment Operators </title>
  <meta charset="utf-8">
  <script>
    var length = 100;
    length++;
    length += 1;
    console.log("Length after += 1: " + length);
    length--;
    length -= 1;
    console.log("Length after -= 1: " + length);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click [Preview](#) . Compare the results to the previous listing results. They should be the same, because we're adding 1 and subtracting 1 using the += and -= operators. So, length++; and length += 1; both yield the same result: adding one to the value of length.

What happens if you try numbers other than 1? Try that now. Make sure you get the results you expect.

Here's a list of assignment operators you can use as a reference for later:

Assignment Operator	Equivalent Expression
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

Comparison Operators

Comparison operators are used to compare values. You can use comparison operators to generate a boolean value (true or false), like this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Comparison Operators </title>
  <meta charset="utf-8">
  <script>
    var frankAge = 14;
    var jimAge = 16;
    var isFrankOlder = frankAge > jimAge;
    console.log("Frank is older? " + isFrankOlder);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **comparison.html** and click [Preview](#) . What value do you see for **isFrankOlder**? Is Frank older than Jim?

We assign the result of the expression **frankAge > jimAge**, which says "is the value of frankAge *greater than* the value of jimAge?", to the variable **isFrankOlder**. In this case, the variable **isFrankOlder** contains the value **false**, because the value of frankAge is *not* greater than the value of jimAge.

The most common way you'll use conditional operators is in conditional expressions in **if statements** (which we'll get to later in this lesson) and in loops.

Here's a list of conditional expressions you can use as a reference for later:

Comparison	Operator	Example
equal	==	var x = 3 var y = 4 x == y returns false
not equal	!=	var x = 3 var y = 4 x != y returns true
equal value and equal type	===	var x = 3 x === 3 returns true x === "3" returns false
greater than	>	var x = 3 var y = 4 x > y returns false
less than	<	var x = 3 var y = 4 x < y returns true
greater than or equal to	>=	var x = 3 var y = 4 x >= y returns false
less than or equal to	<=	var x = 3 var y = 4 x <= y returns true


Conditional Expressions and Statements

You know that a conditional expression is one that results in a boolean value, often by including a comparison operator. For example, **3 > 4** and **(4/2) == 2** are conditional expressions.

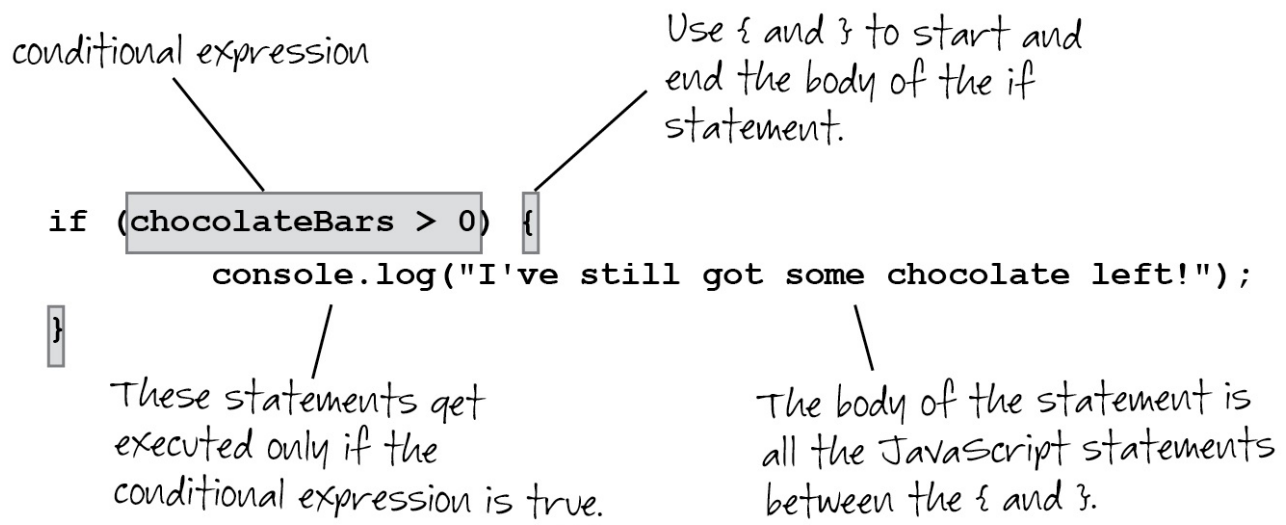
You can use conditional expressions in *conditional statements* to control the flow of your code. So you can say, "If such-and-such is true, then do this, else do that." Let's take a look at an example:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Conditional Statements </title>
  <meta charset="utf-8">
  <script>
    var chocolateBars = 3;
    if (chocolateBars > 0) {
      console.log("I've still got some chocolate left!");
    }
    else {
      console.log("Oh no! I'm out of chocolate!");
    }
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **conditional.html** and click **Preview** . You see the message, "I've still got some chocolate left!" because the value of the variable **chocolateBars** is 3, so the result of the conditional expression, **chocolateBars > 0** is true. That means the statements in the first part of the if statement will run. Try changing the value of **chocolateBars** to 0 and see what happens.

You can use the **if statement** with or without the else. Here's a closer look at the various parts of the if statement with the else, and without it:



The if/else statement has two parts, the if part and the else part.

```
if (chocolateBars > 0) {  
    console.log("I've still got some chocolate left!");  
}  
else {  
    console.log("Oh no! I'm out of chocolate!");  
}
```


The else part is executed only if the conditional expression is false.

Statement nesting

You can make JavaScript statements more complex by *nesting* them inside other statements.

CODE TO TYPE:

```
<!doctype html>  
<html lang="en">  
<head>  
  <title> Conditional Statements </title>  
  <meta charset="utf-8">  
  <script>  
    var chocolateBars = 3;  
    var hungry = true;  
    if (chocolateBars > 0) {  
      console.log("I've still got some chocolate left!");  
      if (hungry == true) {  
        console.log("Eat a chocolate bar!");  
        chocolateBars--;  
      }  
    }  
    else {  
      console.log("Oh no! I'm out of chocolate!");  
    }  
  </script>  
</head>  
<body>  
</body>  
</html>
```

Save your file and click . What console messages do you see? What's the value of **chocolateBars** after the code has run?

The String Concatenation Operator

One last operator before you're done with this lesson: the *string concatenation* operator. "Concatenation" means you take two strings and stick them together, so you could, for instance, combine the strings "chocolate" and "bar" into the string "chocolatebar". You can include spaces, so you could combine a first name, "Jim", with a space, " ", and a last name, "Smith" to create "Jim Smith."

The string concatenation operator is **+**. But wait, you say! Isn't that the arithmetic operator for adding two numbers together?


It is *both*. This is an example of an "overloaded" operator. In most circumstances, it's fairly easy for JavaScript to figure out which operator you mean. So, for example, if you type **var x = 3 + 4**; JavaScript knows you mean to add two numbers together. But if you type **var fullName = "Jim" + " " + "Smith"**; then JavaScript knows you mean to concatenate strings together.

Try these examples now, in the JavaScript console or in a JavaScript program in an HTML file:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Concatenate strings </title>
  <meta charset="utf-8">
  <script>
    var x = 3 + 4;
    console.log(x);

    var fullName = "Jim" + " " + "Smith";
    console.log(fullName);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click **Preview** . Do you see what you expected in the console?

Now try this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Concatenate strings </title>
  <meta charset="utf-8">
  <script>
    var stringOrNumber = "3" + "2";
    console.log(stringOrNumber);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click **Preview** . What do you see in the console?

Now change the code just a little:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Concatenate strings </title>
  <meta charset="utf-8">
  <script>
    var stringOrNumber = 3 + "2";
    console.log(stringOrNumber);
  </script>
</head>
<body>
</body>
</html>
```

Now what do you see in the console? Do you know why?

In the first example, `"3" + "2"`, the double quote s make it pretty clear that we've got two strings we're concatenating together. The second example, however, isn't so clear. In this case, JavaScript assumes you mean to concatenate strings together and converts the number `3` into a string `"3"`, and then concatenates it with the string `"2"`. Typically if you are using `+` with a string, JavaScript will assume you mean concatenation. What happens when you try `3 * "2"`?


Comments

One more quick thing before we close up this lesson. As you start writing more JavaScript code, it's good to know how to *comment* your code; that is, add lines of documentation right in your code so you can describe what you're doing. This helps both you and others read your code and understand it.

There are two ways to add comments. The first is used to comment out *one* line of code:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Comments </title>
  <meta charset="utf-8">
  <script>
    // This is a comment!
    var stringOrNumber = 3 + "2";
    console.log(stringOrNumber);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . It should run just as before. Any text after the `//` is ignored! Try typing some other comments the console, like this:

```
> // This is a comment!
    var stringOrNumber = 3 + "2";
    console.log(stringOrNumber);
32
< undefined
> stringOrNumber
"32"
> // I'm a comment
undefined
> // I'm another comment
undefined
>
```

Notice that, again, the result of the comment is "undefined," which is just fine. It just means that there's no value resulting from the comment.

The `//` characters in a line indicate that everything following them is a comment, through to the end of the line. So you can write:


`var stringOrNumber = 3 + "2"; // is the value a string or a number?`

And all the text after the `//` is a comment, which means it doesn't get executed.

What if you need more than one line of comments? You can use a *multi-line* comment:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Comments </title>
  <meta charset="utf-8">
  <script>
    /*
      var stringOrNumber = 3 + "2";
      console.log(stringOrNumber);
    */
  </script>
</head>
<body>
</body>
</html>
```

Save it and click **Preview** . Look at the console window. You should see nothing at all! Because both lines of code are commented out, neither of these lines is executed.

You can also use this method to add multi-line documentation:

OBSERVE:

```
/*
 * Add some documentation here.
 * More documentation here.
 *
 */
var stringOrNumber = 3 + "2";
```

Try adding some comments to your JavaScript code. Use them whenever you want to help you remember what you did and why.

You've learned about a lot of operators in this lesson. Operators are an important part of JavaScript and you'll use them a lot as you continue through the course. In the next lesson we'll be using operators quite a bit, so make sure you understand the concepts discussed here before moving on to the next lesson.

You've also learned how to use the JavaScript console. This is an invaluable tool for learning JavaScript and, especially when combined with `console.log`, a great way to debug your JavaScript code if you've got errors. It's common to make typing errors as you learn a new language, so now you have a good way to track those down.

Play with the console, try out the statements, expressions, and operators you learned here. Have fun!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Looping

Lesson Objectives

When you complete this course, you will be able to:

- use *while* loops and *for* loops in JavaScript.
- combine looping statements and conditional statements.
- use loops to generate HTML.

A loop is a section of code you want to execute a specific number of times or while a specified condition is true. To help visualize this, let's consider a real life example.

Suppose you want to eat some M&Ms, and you have a bowl of them on your desk. To eat an M&M, you take one from the bowl and eat it. You want another one, so you take another M&M. You continue taking M&Ms from the bowl until they are all gone. You are *looping*: repeating the process of taking M&Ms from the bowl until there are none left.


You can write programs that loop with JavaScript by using a *while* loop or a *for* loop. Let's take a closer look at both of these kinds of loops.

The While Loop

A *while* loop executes the same section of code while a specified condition is true. So, let's write a program to eat M&Ms *while* there are M&Ms left.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> The While Loop </title>
  <meta charset="utf-8">
  <script>
    var mms = 5;
    while (mms > 0) {
      console.log("Eat an M&M");
      mms--;
    }
    console.log("All out of M&Ms");
  </script>
</head>
<body>
</body>
</html>
```

Save it in your `/javascript1` folder as `mandms.html` and click [Preview](#) . If you open the JavaScript console you should see 5 messages that say "Eat an M&M" and one message that says "All out of M&Ms". (And, just as a reminder, if you want to use `alert` instead of `console.log`, you can!) Let's look at the script:

OBSERVE:

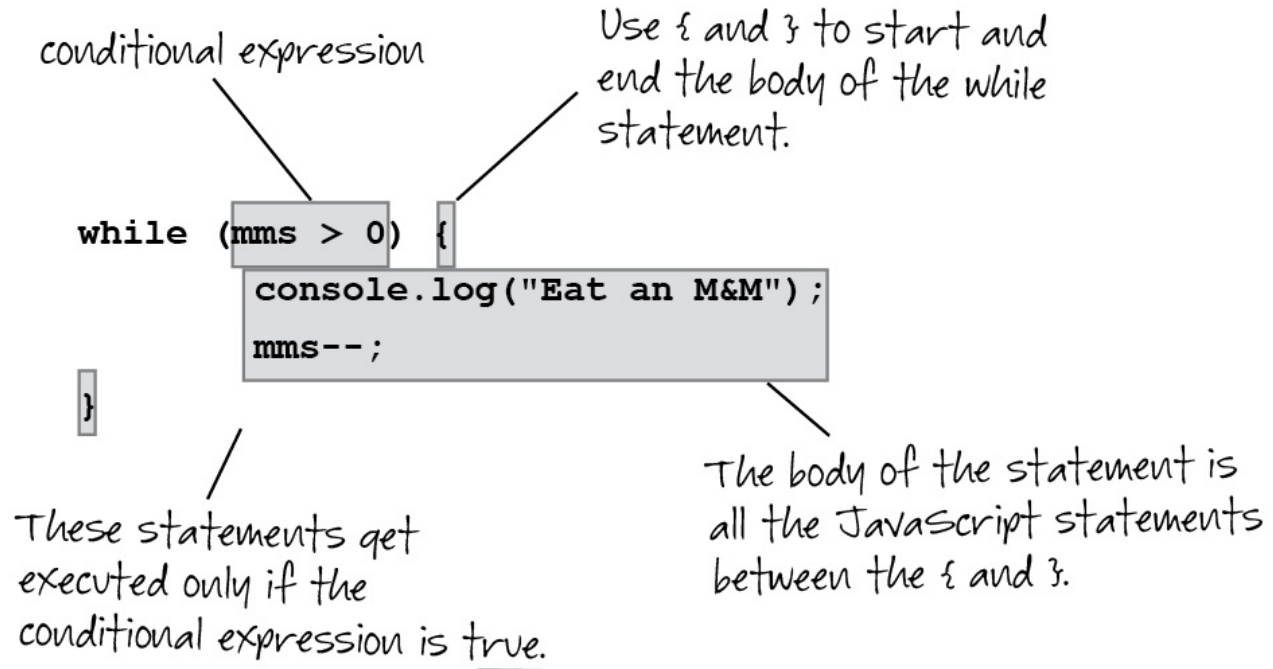
```
<script>
  var mms = 5;
  while (mms > 0) {
    console.log("Eat an M&M");
    mms--;
  }
  console.log("All out of M&Ms");
</script>
```

Notice that we *initialize* the value of the variable `mms` to 5 before the loop starts. This is an important step because `mms > 0` is used in the *conditional test*. If the condition is true, we keep looping and execute the code within the

{braces}.

Also notice that we **decrement** the value of **mms** each time through the loop so the value changes. The first time through the loop, **mms** is 5, the second 4, and so on. The last time through the loop, **mms** is 1 and so this line of code decrements its value to 0. When **mms** is equal to 0, the loop terminates because the value is no longer **greater than 0**.

Here's a review of the different parts of the while statement:



Try experimenting with the example above by using a different initial value, and a different test. For instance, what happens if you write instead:

CODE TO TYPE:

```
<!doctype html>  
<html lang="en">  
<head>  
  <title> The While Loop </title>  
  <meta charset="utf-8">  
  <script>  
    var mms = 0;  
    while (mms < 5) {  
      console.log("Eat an M&M");  
      mms++;  
    }  
    console.log("All out of M&Ms");  
  </script>  
</head>  
<body>  
</body>  
</html>
```


Do you get the same results? Do you see why?

The For Loop

A *for* loop is similar to a *while* loop, but combines the initialization of the loop variable, the conditional test, and modifying the loop variable all into one statement.

CODE TO TYPE:

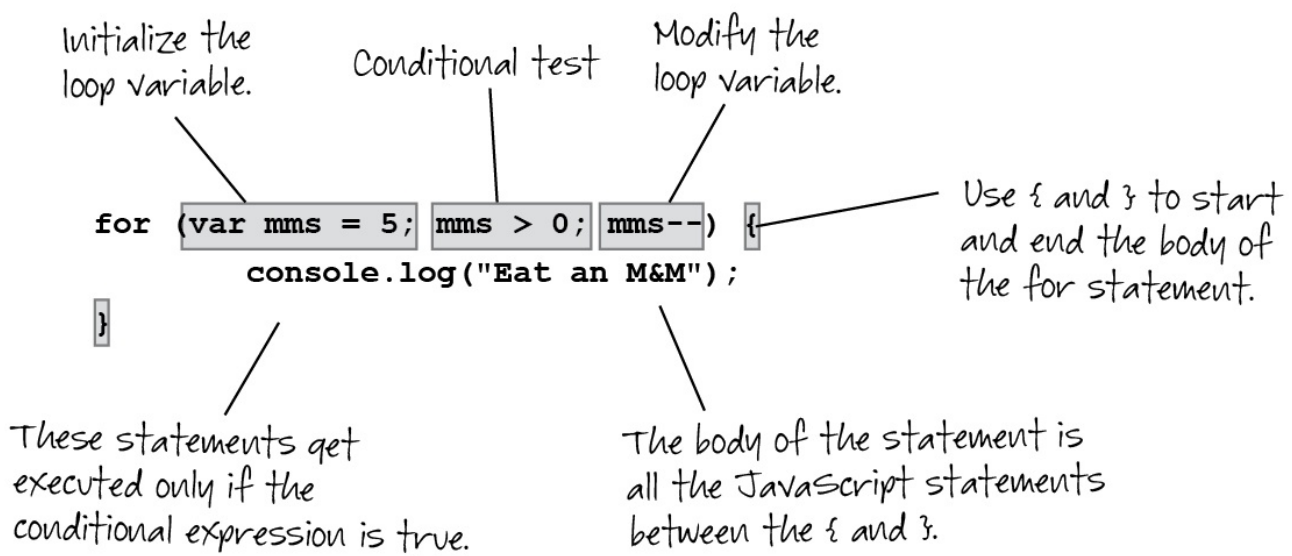
```
<!doctype html>
<html lang="en">
  <head>
    <title> The For Loop </title>
    <meta charset="utf-8">
    <script>
      var mms = 0;
      for (var mms = 5; mms > 0; mms--) {
        console.log("Eat an M&M");
        mms++;
      }
      console.log("All out of M&Ms");
    </script>
  </head>
  <body>
  </body>
</html>
```

Save it and click , and check your JavaScript console. You should see the same messages you saw before (with the while loop). Let's look at the script:

OBSERVE:

```
<script>
  for (var mms = 5; mms > 0; mms--) {
    console.log("Eat an M&M");
  }
  console.log("All out of M&Ms");
</script>
```

Here's how it works: the variable **mms** is set to 5. Then the **conditional test** is evaluated, and if it results true, the **body** of the for loop is executed; that is, all the statements between the **curly braces**. After the body is executed, the variable **mms** is decremented, and the **conditional test** is evaluated again. As long as the **test** evaluates to true, the for loop keeps going. In this example, that happens 5 times, so you see five "Eat an M&M" strings displayed. Once **mms gets to 0**, the loop terminates, and the next statement in the code is executed, which displays "All out of M&Ms".



Can you change the for loop so **mms** starts at 0 and goes to 5 (like we did with the while loop earlier)? See if you can figure out how to do that.

While or For?

In this example, we implemented the exact same loop twice; that is, we can do the same thing with either a while loop or a for loop. So how do you know which kind of loop to use?


Sometimes either one will work, and you can pick whichever you like best. Programmers tend to favor for loops over while loops because they are a bit more concise. However, there may be times when a for loop simply doesn't fit the bill and you can use a while loop instead.

Combining Looping Statements and Conditional Statements

Suppose we want to do something different in the body of the loop depending on where we are in the loop. Maybe we want to check how many M&Ms are left before we say it's okay to eat more. Let's take a look at an example:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
  <head>
    <title> The For Loop Combined with an If Statement </title>
    <meta charset="utf-8">
    <script>
      for (var mms = 5; mms > 0; mms--) {
        if (mms >= 3) {
          console.log("Still lots of M&Ms left, so eat more!");
        } else {
          console.log("Getting low on M&Ms, take it easy!");
        }
      }
      console.log("All out of M&Ms");
    </script>
  </head>
  <body>
  </body>
</html>
```

Save it and click  and check the console. In this version, you should see three of the "Still lots of M&Ms left, so eat more!" message, two of the "Getting low on M&Ms, take it easy!" message, and one of the "All out of M&Ms" message.

Here we've combined a for loop with an if statement. The for loop executes 5 times, and each time through the loop, the if statement executes and tests to see if the value of `mms` is greater than or equal to 3. If it is, we display one message in the console; if it's less than 3, we display another. So you'll see a total of 5 messages, one for each time through the loop.

You can combine JavaScript statements like this together to give you lots of options in creating behavior in your programs. Try some of your own combinations now.

BONUS: Using Loops to generate HTML

You've been very patient as we've been writing small programs to write messages to the console, but if you're anything like me, you're probably itching to write some code to actually do something with a real web page. Right?

We've still got a ways to go before we get to *really* working with web pages, but here's a taste.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Output to a Page </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;
    function init() {
      var output = "";
      for (var mms = 5; mms > 0; mms--) {
        if (mms >= 3) {
          output += "Still lots of M&Ms left, so eat more!<br>";
        } else {
          output += "Getting low on M&Ms, take it easy!<br>";
        }
      }
      output += "All out of M&Ms";
      var p = document.getElementById("output");
      p.innerHTML = output;
    }
  </script>
</head>
<body>
  <p id="output">
  </p>
</body>
</html>
```

Don't worry if you don't understand all this code just yet. We'll get to all of it in the next few lessons. In the meantime, click [Preview](#). Instead of seeing messages about the status of your M&M stash in the console, you should now see the messages in the actual web page.

□

OBSERVE:

```
<script>
  window.onload = init;
  function init() {
    var output = "";
    for (var mms = 5; mms > 0; mms--) {
      if (mms >= 3) {
        output += "Still lots of M&Ms left, so eat more!<br>";
      } else {
        output += "Getting low on M&Ms, take it easy!<br>";
      }
    }
    output += "All out of M&Ms";
    var p = document.getElementById("output");
    p.innerHTML = output;
  }
</script>
</head>
<body>
  <p id="output">
  </p>
</body>
</html>
```

Here, we build up a string, **output**, with the messages as we loop. Remember that the operator **+=** is like saying "add this new value to the current value of the variable." (In this case "add" means "concatenate" because the variable is a string). So, it's like writing **output = output + "Still lots of M&Ms left, so eat more!
"**.

Also notice a couple of things we changed about the string. First, we are using the HTML entity **&M** to represent an ampersand. The reason we need to do this is that HTML sees **&** as a special character. Now most browsers will work

even if you don't use `&`, but it's a best practice to always use it. Also notice that we're adding a `
` to the end of each line in the loop, so when we display the lines within the `paragraph` in the body of the page, they include those newlines at the end.

Once the loop has terminated, we add one more string on the end. Then we do a bit of magic to get the `element` where we want to add the string. We're using a function `document.getElementById()`, which retrieves an element using its `id` (notice the `id` attribute on the `<p>` element in the body of the HTML). Once we have the `<p>` element stashed in the variable `p`, we can use the `innerHTML` property of the element to set the content of the `<p>` element to the string that's stored in the variable `output`. Doing this dynamically updates the web page, and you see the string.

We wrap the whole thing in a function named `init`. Why are we doing that? Because we don't want to run the code until the page has loaded. To do that, we set the property `window.onload` to the function *name*, `init`. This tells the browser, "Wait until the page loads before you call the `init` function." You'll get a lot more detail about the reason why this is important later, but for now, just know that if you're updating a web page, it's important to *wait* until the page has been fully loaded by the browser before you update it via JavaScript code that's in the head of your page.

So now you know just a little about how to use JavaScript variables, operators, conditional statements, and loops to update a web page. There will be lots more of that type of code to come, promise!

Loops are powerful tools, and you will use them often in your scripts. Make sure you experiment with them a bit more before moving on to the next lesson. See you there!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Arrays

Lesson Objectives

When you complete this course, you will be able to:

- create arrays and access their values.
- change the values in an array.
- loop with arrays.
- process arrays in JavaScript.

Creating Arrays and Accessing Their Values

An *array* is a collection of values. Think of it like a filing cabinet. Each value in the array gets its own spot, which is accessible by a number, or *index*. Let's take a look at an example:

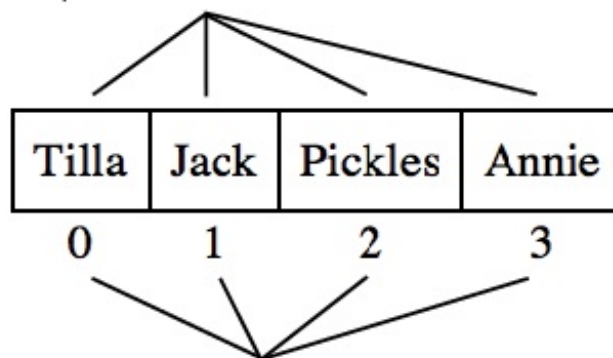
OBSERVE:

```
var pets = ["Tilla", "Jack", "Pickles", "Annie"];
```

We use the `[` and `]` characters to collect the items together and tell JavaScript that we've got an array.

You can think of the array as looking like this:

An array is a collection of values.



Each value in the array has an index.

Each value in the array has an *index* that you can use to access it. Say you want to access "Pickles." "Pickles" is the value in the 3rd spot in the array, but has the index **2** because array indices start with 0. So, you would write:

OBSERVE:

```
var petName = pets[2];
```

and the value of the variable **petName** will be "Pickles."

You can also store numbers in an array:

OBSERVE:


```
var petsAges = [8, 4, 6, 7];
```

To access the number values in an array, you also use an index. So to access the age of the first pet you'd write `var petAge = petsAges[0]`;

Let's create an array and test it out a bit.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var pets = ["Tilla", "Jack", "Pickles", "Annie"];
    console.log("First pet: " + pets[0]);
    console.log("Second pet: " + pets[1]);
    console.log("Third pet: " + pets[2]);
    console.log("Fourth pet: " + pets[3]);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **arraytest.html** and click **Preview** . You should see console messages with the pets names displayed, in the order you access them. Try switching the order around. Try accessing **pets[4]**—because the array has only four values, with indices 0 - 3, if you try to access a value at an index that doesn't exist, you get an **undefined** message.

Try creating another array now, with values of your choice. Can you put both strings and numbers into the same array? Is this a good idea? Why or why not?

Changing the Values in an Array

Once you've created an array, the values are not set in stone. Just like any other variable, you can change the values in an array. Let's start by creating an empty array, and set some values, and then change them.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var icecream = [];
    icecream[0] = "Vanilla";
    icecream[1] = "Chocolate";
    icecream[2] = "Strawberry";
    console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
    icecream[1] = "Chocolate Chip";
    console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click **Preview**  and check the console. You should see:

```

> var icecream = [];
   icecream[0] = "Vanilla";
   icecream[1] = "Chocolate";
   icecream[2] = "Strawberry";
   console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
   icecream[1] = "Chocolate Chip";
   console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
Flavors: Vanilla, Chocolate, Strawberry
Flavors: Vanilla, Chocolate Chip, Strawberry
< undefined
>

```

Let's take a closer look:

OBSERVE:

```

<script>
  var icecream = [];
  icecream[0] = "Vanilla";
  icecream[1] = "Chocolate";
  icecream[2] = "Strawberry";
  console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
  icecream[1] = "Chocolate Chip";
  console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
</script>

```

We create the array by setting the variable **icecream** to an empty array, **[]**. Then, we **add items to the array** one at a time. After displaying those items in the console, we then **change one of the items**, the second item in the array (with the index 1), by setting it to a new value.

Here's something to try:

CODE TO TYPE:

```

<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var icecream = [];
    icecream[0] = "Vanilla";
    icecream[1] = "Chocolate";
    icecream[2] = "Strawberry";
    console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
    icecream[1] = "Chocolate Chip";
    icecream[4] = "Pistachio";
    console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2] + "
, " + icecream[4]);
    console.log("Missing flavor: " + icecream[3]);
  </script>
</head>
<body>
</body>
</html>

```

(Notice that we skipped **icecream[3]**!) Save it and click **Preview** and check the console.

It displays **undefined**. We've seen this before; it just means that a variable (in this case, an item in an array) hasn't been defined yet, or that the result of an expression or statement is undefined (that is, has no specific value).

And, what this means is that you can have holes in your arrays! These are no big deal as long as you know this can happen and write your code accordingly.

Try changing the values of **icecream** to your favorite flavors and displaying them in the console (or with **alert**). Also,

try this: at the console prompt, type:

INTERACTIVE SESSION:

```
> icecream
```

(Just type the text in blue; the > is the prompt).

The console should display the entire array. Here's the result of the above code in the console, and the result of typing **icecream** at the console:


```
> var icecream = [];
    icecream[0] = "Vanilla";
    icecream[1] = "Chocolate";
    icecream[2] = "Strawberry";
    console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2]);
    icecream[1] = "Chocolate Chip";
    icecream[4] = "Pistachio";
    console.log("Flavors: " + icecream[0] + ", " + icecream[1] + ", " + icecream[2] + ", " + icecream[4]);
    console.log("Missing flavor: " + icecream[3]);
Flavors: Vanilla, Chocolate, Strawberry
Flavors: Vanilla, Chocolate Chip, Strawberry, Pistachio
Missing flavor: undefined
< undefined
> icecream
["Vanilla", "Chocolate Chip", "Strawberry", undefined, "Pistachio"]
>
```

Looping with Arrays

In the previous lesson you learned about **while** and **for** loops; you'll find you often use loops with arrays, because loops give you an easy way to access every element in an array. Let's take a look.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var icecream = ["Vanilla", "Chocolate Chip", "Strawberry"];
    for (var i = 0; i < icecream.length; i++) {
      console.log("Flavor: " + icecream[i]);
    }
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . All the ice cream flavors display in the console.

OBSERVE:

```
<script>
  var icecream = ["Vanilla", "Chocolate Chip", "Strawberry"];
  for (var i = 0; i < icecream.length; i++) {
    console.log("Flavor: " + icecream[i]);
  }
</script>
```

First, notice that in the **for** loop, we use the **length** of the array as a way to know when to stop looping. JavaScript arrays have a **length** property that gives you the length of the array. In this case, the length of the array **icecream** is 3,

so the loop values for **i** will be 0, 1, and 2, successively.


Next, notice that we're using **i as the index** to access the values in the array as we loop. So, the first time through the loop, when **i** is 0, we'll display the value of **icecream[i]** or **icecream[0]**, which is "Vanilla." The second time through, **i** is 1, so we display the value of **icecream[1]**, which is "Chocolate Chip." And so on, until the loop terminates when **i** is 3.

Is there something interesting about the **length** property you might have noticed? The **length** of an array is always one greater than the last index. So if an array has three items, the **length** is three, but the last index is 2 (because we start counting indices at 0).

You can rewrite this loop, using a while loop instead, like this:

CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var icecream = ["Vanilla", "Chocolate Chip", "Strawberry"];
    var i = 0;
    while (i < icecream.length) {
      console.log("Flavor: " + icecream[i]);
      i++;
    }
  </script>
</head>
<body>
</body>
</html>
```

Save it and click [Preview](#) . You should see the exact same results. Go back to the previous lesson if you need a reminder on how a while loop works vs. a for loop. Remember, with a while loop, it's really important to remember to *change the loop variable in the loop*—otherwise you might get an *infinite loop*!

If you create an infinite loop by mistake, the best way to get out of it is to just close the browser window. Try it:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var icecream = ["Vanilla", "Chocolate Chip", "Strawberry"];
    var i = 0;
    while (i < icecream.length) {
      console.log("Flavor: " + icecream[i]);
      i++;
    }
  </script>
</head>
<body>
</body>
</html>
```

If we don't increment the loop variable **i**, it will *always* be less than **icecream.length**. Save it and click [Preview](#) . Open the console window right away. You'll see console messages about the flavor of the first icecream in the array flying by. To get out of the loop, just close the browser window.

Processing Arrays

Arrays pop up fairly often in JavaScript—whether you need to create them yourself to store data, or to access and


process built-in arrays, like arrays of HTML elements (you'll find out how to get arrays of HTML elements and process them later in the course!). Let's do a couple of array processing examples so you can get more practice with them.

Creating an Array from a String with Split

Suppose you want to *parse*, or split up, a string; for example, breaking a sentence into separate words? We can do that!

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var quote = "Common sense and a sense of humor are the same thing, moving at
different speeds. A sense of humor is just common sense, dancing."
    var quoteArray = quote.split(" ");
    for (var i = 0; i < quoteArray.length; i++) {
      console.log(i + ": " + quoteArray[i]);
    }
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . After splitting the array, all the words are in separate array items. We're displaying the results in the console by looping through the array, and for each item in the array, displaying its index and the value of the array at that index. Notice anything about the items at indices 10, 14, 22 and 23?

(That quote is from William James, in case you're curious). Our code turns this string into an array of strings with the **split()** function:

OBSERVE:

```
var quote = "Common sense and a sense of humor are the same thing, moving at dif
ferent speeds. A sense of humor is just common sense, dancing."
var quoteArray = quote.split(" ");
```


This splits the string **quote** into separate strings whenever it finds a space (" ").

Searching for a String

Now that we have an array of words, let's search the array for a specific word, "of."

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var quote = "Common sense and a sense of humor are the same thing, moving at
different speeds. A sense of humor is just common sense, dancing."
    var quoteArray = quote.split(" ");
    for (var i = 0; i < quoteArray.length; i++) {
      console.log(i + ": " + quoteArray[i]);
      if (quoteArray[i] == "of") {
        console.log("Found an of");
      }
    }
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . How many "ofs" did you find?

Working with Numbers

Finally, let's make a program add 1 to each number in an array.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Arrays </title>
  <meta charset="utf-8">
  <script>
    var scoopsOfIcecream = [3, 2, 4, 5, 2];
    for (var i = 0; i < scoopsOfIcecream.length; i++) {
      var scoops = scoopsOfIcecream[i];
      scoopsOfIcecream[i] = scoops+1;
    }
    console.log(scoopsOfIcecream);
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . Do you see the new values in the array displayed in the console?

OBSERVE:

```
<script>
  var scoopsOfIcecream = [3, 2, 4, 5, 2];
  for (var i = 0; i < scoopsOfIcecream.length; i++) {
    var scoops = scoopsOfIcecream[i];
    scoopsOfIcecream[i] = scoops+1;
  }
  console.log(scoopsOfIcecream);
</script>
```

We loop through the array in the same way we have done. In the body of the loop, we **get the value from the array at index i, and save it in a variable, scoops**. We then **increase the value of scoops and store the new value back in the array at the same index**.

This is a little tricky, so let's step through it. The first time through the loop, `i` is 0, so we get the value of the first item in the array, `scoopsOfIceCream[0]`, which is 3. We store that value, 3, in the variable `scoops`.

Then, we set the value of `scoopsOfIceCream[0]` to the value of `scoops+1`, which is 4. So now, the values in the array are `[4, 2, 4, 5, 2]`.

We do this for each value in the array, so when the loop ends, the values in the array are `[4, 3, 5, 6, 3]`.

You'll see many more arrays in the course. Spend some time practicing a bit before you do the project for this lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

The Document Object Model

Lesson Objectives

When you complete this course, you will be able to:

- use an internal representation of the page called the Document Object Model.
- use document to access and update parts of your page.
- access and update elements.
- use the window object.

So far in this course, we've been writing JavaScript and seeing the results using (mostly) **alert** and **console.log**, or typing JavaScript directly into the console.

However, the real goal of learning JavaScript is to write code that interacts with a web page. To do that, you need to know a bit more about how the browser represents a web page, and understand some of the built-in JavaScript objects, functions, and properties that you'll use to interact with the page.


Behind the Scenes of a Web Page

When you load a web page into your browser, either by clicking a link or by typing in a URL and pressing the Enter key, your browser retrieves the page, and begins to *parse* the page. Parsing just means the browser takes the HTML text and turns it into an *internal representation* of the page. That internal representation is called the *Document Object Model*.

The Document Object Model, often just called the DOM, is a hierarchical collection of objects that represent the page and the objects in it. Let's take a look at a simple example:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
  <head>
    <title> Simple DOM </title>
    <meta charset="utf-8">
  </head>
  <body>
    <p id="answer">
      The answer to life, the universe and everything is 42!
    </p>
  </body>
</html>
```

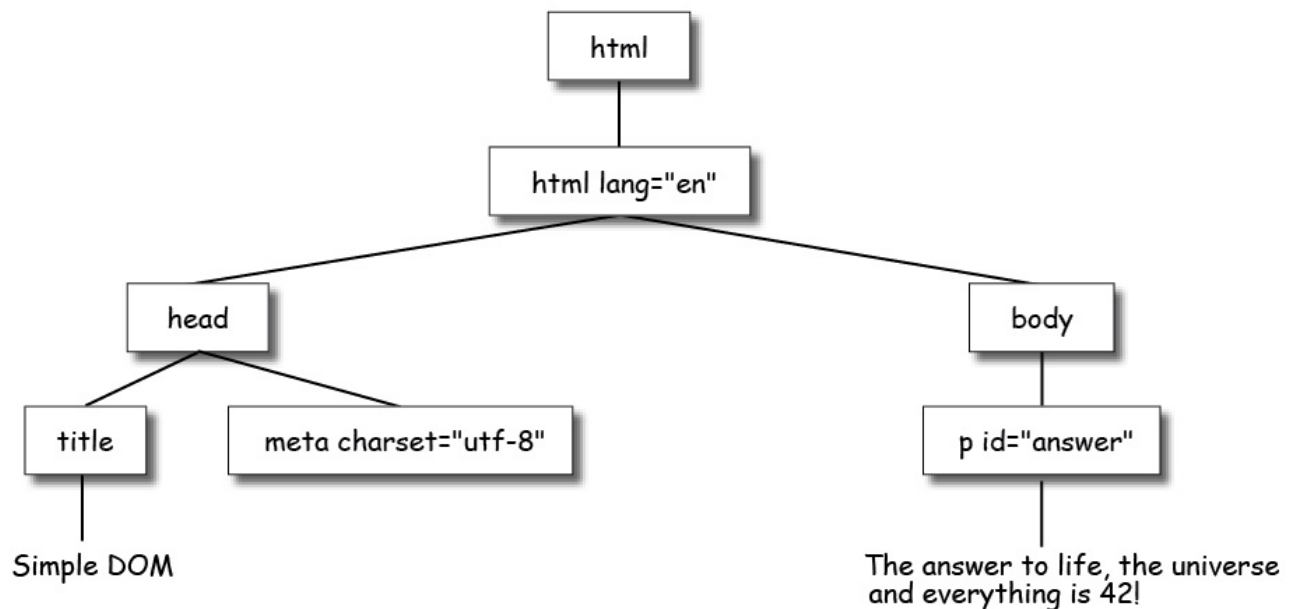
Save it in your **/javascript 1** folder as **dom.html** and click **Preview** . Now, we'll use the browser's developer tools to look at the page "behind the scenes." Watch [this video](#) on how to access the developer tools to see the elements in your page:

(For Internet Explorer 9, see [this video](#).)

Now try it yourself with the example above. If you need to review how to access the developer tools in your browser of choice, review them [here](#). Then, add some elements to this simple example, reload the page, and again view the elements with the developer tool.

The Document Object Model and document.getElementById()

The browser represents the DOM elements in your page as hierarchically nested objects, and we usually visualize it as an upside-down tree:



If you turn this image upside down, you'll see the "tree": **html** is the *root* of the tree, and everything in the page "grows" from the root. Notice that the *nesting* of the elements in your page corresponds to the grouping of elements in the hierarchy of the tree. So, for example, the **<title>** and **<meta>** elements are nested inside the **<head>** element in your HTML, so the title and meta objects are grouped beneath the head object in the DOM tree.

Notice that I've included the attributes of the elements in the tree, like the **charset** attribute of the **<meta>** element, and the **id** attribute of the **<p>** element. I've included those because they get included in the internal representation of the page, and you can access those pieces through code too.

Finally, notice that I also included the *text content* of the elements, like the **<title>** and the **<p>**. You can access the content of any element, and update it too.

You can get access to this tree using the **document** object. **document** is a built-in JavaScript object that gives you access to the elements in your page through JavaScript code. You'll learn more about objects later in the course; for now, think of an object as a collection of properties and functions. You can use the document object to access properties of your page, and you can use the document object's functions—called **methods** because they are functions associated with an object, rather than independent functions you wrote yourself—to perform actions on the page.


Don't worry too much about what objects and functions and methods are at this point in the course; for now, know that objects are a way to collect properties together into one variable, and functions and methods are how we collect code together into reusable chunks that we can call whenever we want to execute that code again. The only difference between a function and a method is that a method is a function associated with an object.

Let's take a look at some of the ways you can use **document** to access and update parts of your page.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Simple DOM </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var p = document.getElementById("answer");
      var answer = prompt("Enter your answer to life, the universe and everything:");
      p.innerHTML = answer;
    }
  </script>
</head>
<body>
  <p id="question">
    What is the answer to life, the universe and everything?
  </p>
  <p id="answer">
    The answer to life, the universe and everything is 42!
  </p>
</body>
</html>
```

Save it and click . Enter an answer when prompted. When you click **OK**, the answer is added to the web page. Now let's take a closer look.

OBSERVE:

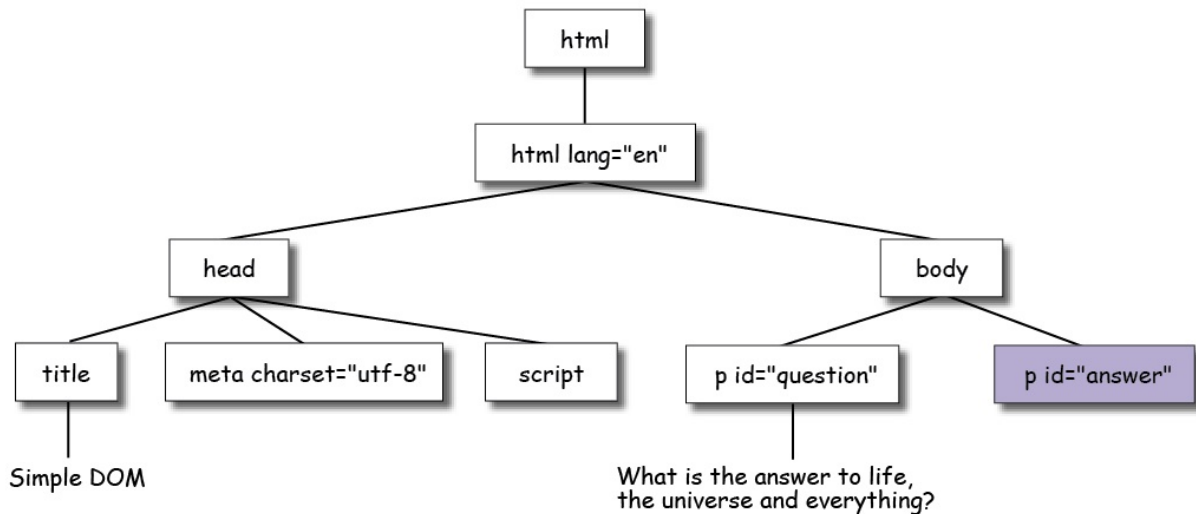
```
<script>
  window.onload = init;

  function init() {
    var p = document.getElementById("answer");
    var answer = prompt("Enter your answer to life, the universe and everything:");
    p.innerHTML = answer;
  }
</script>
```

There are two steps in this code where your JavaScript interacts directly with the web page: first, where you **get the <p>** element from the page, and second, where you **update the content of the <p> element**.

Getting Access to Elements

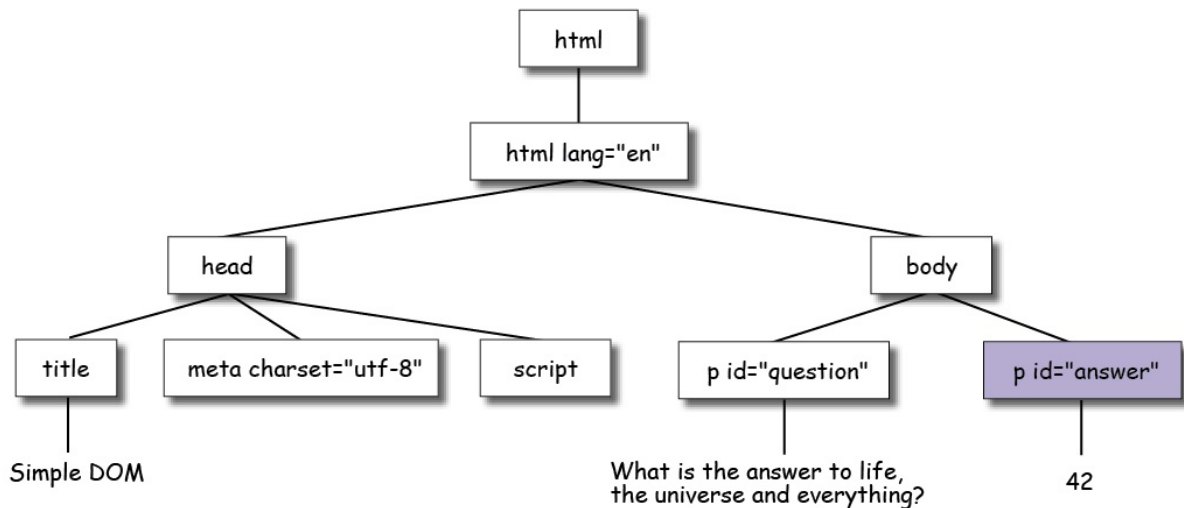
So what's really going on with `document.getElementById("answer")`? Let's break it down into three parts. We use the `document` object's method `getElementById` to get the element with the id `"answer"`. A method is just a function associated with an object. What does it mean to "get an element"? It means we're getting a handle to the *element object* in the DOM tree that represents that element in the HTML. In this case, we get the object that represents the `<p>` element with the id `"answer"`. Once we have that element object, we can access its properties and methods.



We have a new element object, **script**, grouped under the **head** and we have two **p** element objects in the **body**, each with its own id. By using the id "answer" in `document.getElementById("answer")`, we target one specific **p** object.

Updating Elements

One of the properties of an element object is the **innerHTML** property. This represents all the content of that element. This could include additional HTML elements, or just text, as in this example. We're using this property to set the content of the **<p> element with the id "answer"** by setting **p.innerHTML** to the content of the variable **answer**, which contains the value you typed in at the prompt. If you typed "42," the DOM tree looks like this after this JavaScript code runs:



This is really cool, because you are updating your web page dynamically using JavaScript! The web page will magically change to include the new content when this code runs.

So, `document.getElementById()` is a common way you'll get access to elements in your page using JavaScript, and the **innerHTML** property is a common way you'll update the content of elements. These are the most useful tools in your toolbox for using JavaScript to interact with your page, but there are many more.

Getting Elements with document.getElementsByTagName()


With `document.getElementById()`, you can retrieve only one element at a time from the DOM. Why? Because, remember that in HTML, ids must be unique, so only one element can have a given id. This is a good thing because we want to be able to target elements uniquely, using ids with both CSS and JavaScript.

However, there are many times when you want to access multiple elements at once. In this case, it's more efficient to access elements another way; for example, using the elements' tag name instead. Let's take a look at an example. Create a new file as shown:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Accessing multiple elements </title>
  <meta charset="utf-8">
  <script>

  </script>
</head>
<body>
  <form>
    <p>Enter numbers to add:</p>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> <br>
    <p>= <span id="sum"></span></p>
    <input type="button" id="submit" value="Add">
  </form>
</body>
</html>
```

Save it in your **/javascript 1** folder as **dommulti.html** and click . There's no script yet to process the input, so the **Add** button doesn't do anything.

Let's step through the HTML for the input form:

OBSERVE:

```
<form>
  <p>Enter numbers to add:</p>
  <input type="number" size="4"> + <br>
  <input type="number" size="4"> + <br>
  <input type="number" size="4"> <br>
  <p>= <span id="sum"></span></p>
  <input type="button" id="submit" value="Add">
</form>
```

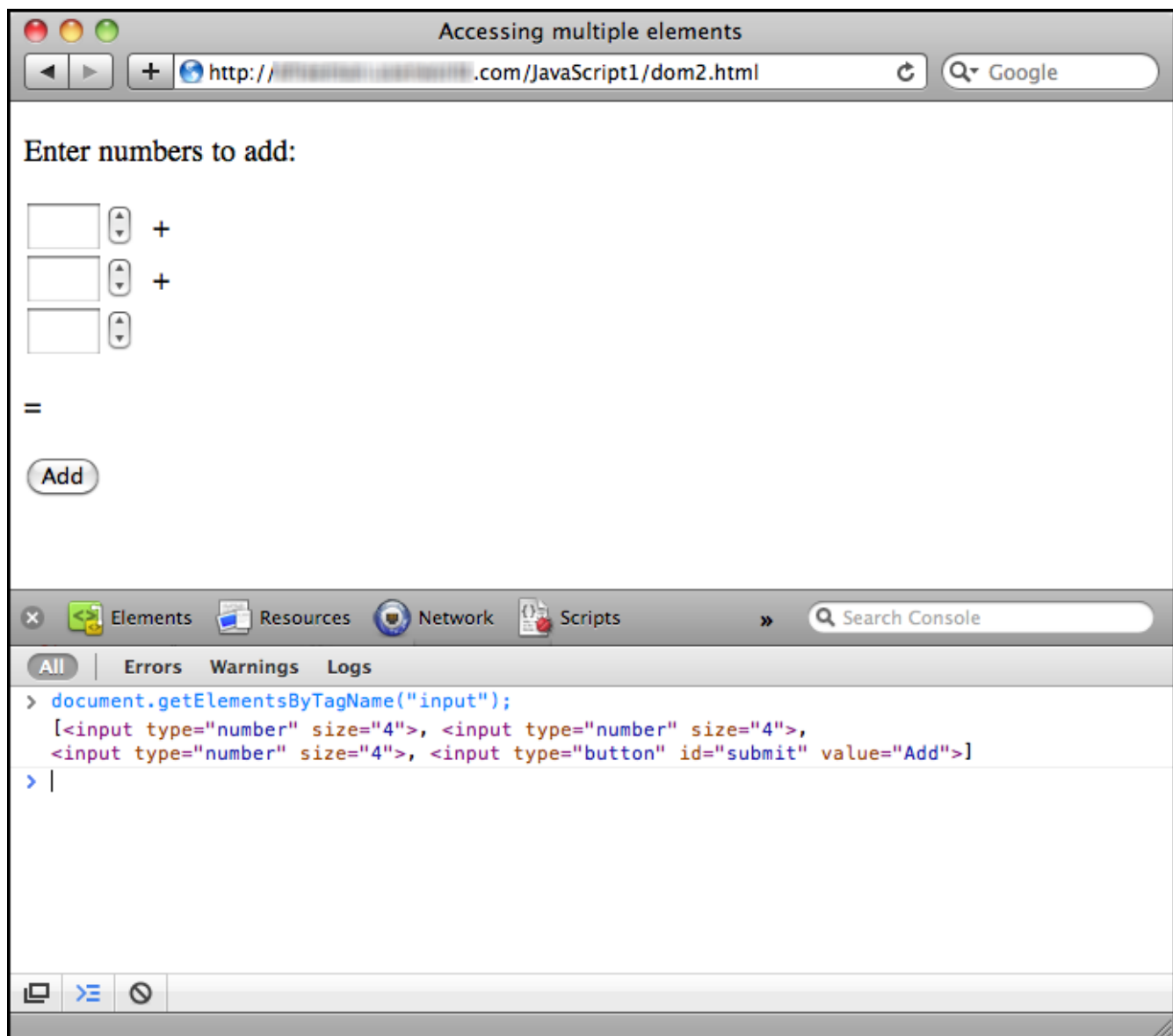
We have a form with **three "number" <input> elements**, and a **submit button <input> element**. When the user clicks the **Add button**, we want to add up all the numbers entered in the **number inputs**, and put the resulting sum in the ** element with the id "sum"**.

Open the JavaScript console in your browser in the window where you've previewed the HTML above, and enter the command as shown:

TYPE IN THE CONSOLE:

```
document.getElementsByTagName("input");
```

You'll see the four **<input>** elements in the page: the three number inputs, and the button input. It looks something like this:



So, **document.getElementsByTagName()** returns a collection of HTML elements—all the elements that match the id you specify in the call to **document.getElementsByTagName()**. What if you want to use **document.getElementsByTagName()** in a program? And once you've got the collection of elements that is the result of calling **document.getElementsByTagName()**, what can you do with those elements? Let's find out. We'll write a JavaScript program to take the values the user types into the form inputs, add them up, and put the result in the page in the "sum" **** element.

Setting Up the window.onload and button.onclick Events

You might remember from an earlier lesson when we said that you can't access values in the DOM—either to read or update—before the page has completely loaded. That means we can't call **document.getElementsByTagName()** until the page has loaded. So we're going to need to set up a function to call when the page has loaded, by setting the **window.onload** property, like we did in the previous example above.

But we don't want to run the JavaScript code to add up the numbers when the *page* loads; we want to add them up only when you *click the Add button*. Just like we can delay executing code until after the page has loaded with the **window.onload** property, we can also delay executing code until the user clicks a button with the **button.onclick** property.


So, how do you tell JavaScript to run code when a button is clicked? Let's take a look:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Accessing multiple elements </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var button = document.getElementById("submit");
      button.onclick = addUp;
    }

    function addUp() {
      alert("testing sum");
    }
  </script>
</head>
<body>
  <form>
    <p>Enter numbers to add:</p>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> <br>
    <p>= <span id="sum"></span></p>
    <input type="button" id="submit" value="Add">
  </form>
</body>
</html>
```

Save it and click . Try clicking the **Add** button. Do you see the alert?

Now, again, don't worry too much if you don't understand all this code yet; you'll learn more about functions and events shortly. Get the basic idea for how this works, and then focus on how we used JavaScript to get and update elements (the next step).

OBSERVE:

```
<script>
  window.onload = init;

  function init() {
    var button = document.getElementById("submit");
    button.onclick = addUp;
  }

  function addUp() {
    alert("testing sum");
  }
</script>
```

First we set up an **init()** function that will execute when the page finishes loading. By setting the **window.onload** property to **init**, we're telling JavaScript, "Run the function **init** when you finish loading the page."

The **init** function gets the button element from the DOM using **document.getElementById()** and its id **"submit"** (look in the form HTML for the button, and you'll see it has that id).

Now, just like we can set the **window.onload** property to a function to tell it to run that function when the page is loaded, we can assign a function to a button element's *onclick property* to tell it to run that function when the button is clicked. So by setting **button.onclick**, we're telling JavaScript to call the **addUp()** function when the Add button is clicked. This is called "event handling" and we'll get into a lot more detail in the next lesson!

In the **addUp()** function, so far, we merely added an alert so we can see the button is working. Now we need to actually add up the numbers!

Get the Form Input Values and Update the Page


The next step is to write the code for the **addUp()** function. This function will get all the values typed into the form inputs, add them up, and put the result in the "sum" ****.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Accessing multiple elements </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var button = document.getElementById("submit");
      button.onclick = addUp;
    }

    function addUp() {
      alert("testing sum");
      var sum = 0;
      var inputs = document.getElementsByTagName("input");
      for (var i = 0; i < inputs.length - 1; i++) {
        var addendString = inputs[i].value;
        var addend = parseInt(addendString);
        if (!isNaN(addend)) {
          sum += addend;
        }
      }
      var span = document.getElementById("sum");
      span.innerHTML = sum;
    }
  </script>
</head>
<body>
  <form>
    <p>Enter numbers to add:</p>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> <br>
    <p>= <span id="sum"></span></p>
    <input type="button" id="submit" value="Add">
  </form>
</body>
</html>
```

Save it and click **Preview** . Enter some numbers and click **Add**. What happens if you leave one of the form inputs blank? What if you enter a string instead?

Note Some browsers won't let you type a string into a "number" input; others will.

OBSERVE:

```
function addUp() {  
  var sum = 0;  
  var inputs = document.getElementsByTagName("input");  
  for (var i = 0; i < inputs.length - 1; i++) {  
    var addendString = inputs[i].value;  
    var addend = parseInt(addendString);  
    if (!isNaN(addend)) {  
      sum += addend;  
    }  
  }  
  var span = document.getElementById("sum");  
  span.innerHTML = sum;  
}
```

Let's step through this code. First, we initialize the variable `sum` to 0; this is where we'll keep a running total of the numbers in the form.

Next, we use `document.getElementsByTagName("input")` to get all the elements with the tag name "input." This returns four input element objects in an **HTML collection**, as you saw earlier when you used the console. We store that result in the variable `inputs`.

An **HTMLCollection** is similar to an array, and you can use a **for loop** to access each element in the collection just like you use a **for loop** to access each item in an array.

Note

A collection is *not equivalent* to an array, so you can't do everything with the collection that you can with an array. But the syntax for looping through the items in the collection is exactly the same as for looping through an array.

So, we loop through the items in the `inputs` collection so we can get each number from the "number" inputs. But remember, there's one `<input>` element in the collection that's *not* a number: the "button" `<input>` element! So instead of looping four times, we only want to loop three times, so our looping test uses `inputs.length - 1` instead of the usual `inputs.length`. Of course, this only works because we know that the "button" input will be the last item in the array. If that wasn't the case, we'd have to check each item in the collection to make sure it was a "number" input and not a "button" input.

Each item in the collection—that is, each `inputs[i]`—is an input element object. We can get the value of what the user typed into a given input element by using its `value` property. So, for instance, `inputs[i].value` will contain the string "3" if you typed in 3.

Notice that the value we get from the input is a string, not a number! We can convert the string to an integer number with the function `parseInt()`. But what if you type in "x" or "cheese" instead of a number? Then that number isn't going to get converted to a number correctly, right? If the string you pass to the `parseInt` function doesn't represent an actual number, then the result of the function is NaN, which means "Not a Number." So, the variable `addend` will contain NaN and if we add that to `sum`, it will mess everything up (that is, it will cause the entire sum to be NaN!).

So, we need to **check to see if the value in the addend variable is NaN**. To do this, we use the `isNaN()` function. If `addend` is *not* equal to NaN, then we can add it to `sum`. Otherwise, we just ignore it.

Finally, we get the "sum" `` element using `document.getElementById()`, and set its `innerHTML` property to the value in `sum`. This causes the page to update, and the number appears in the page!

That was a lot of new concepts to get through, so spend some time going over this code again, and make sure you understand each part. Can you draw the DOM tree for the HTML in this example? Try drawing the DOM tree for both before and after you click the Add button.

Experiment a little. What happens if you enter a negative number, like -3, in the form? Do you get the correct sum? What about if you enter a floating point number, like 3.2, into the form? Do you get the correct sum? If not, why not? What if you change `parseInt` to `parseFloat`?

The Window Object

So now you know that the **document** object is an important object because it's how you get JavaScript to talk to your web page, access elements and content in the page, and update your page.

You've also been using another important object: the *window* object. You used it when you set the **window.onload** property to delay executing a JavaScript function until after the page loads.

That window is the **default object** in any JavaScript program. What does that mean? It means that anything you do at the "top level"—like using document (for example, with **document.getElementById**) or a function like **alert()**—you're actually doing in the window object!

So instead of **alert("hey");** you could write **window.alert("hey");**. Or instead of **console.log("hey");**, you could write **window.console.log("hey");**. The same is true with document: instead of **document.getElementsByTagName("input");**, you could write **window.document.getElementsByTagName("input");**. You can try it yourself in the JavaScript console window, or by creating an HTML file like this:

CODE TO TYPE:

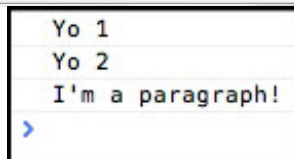
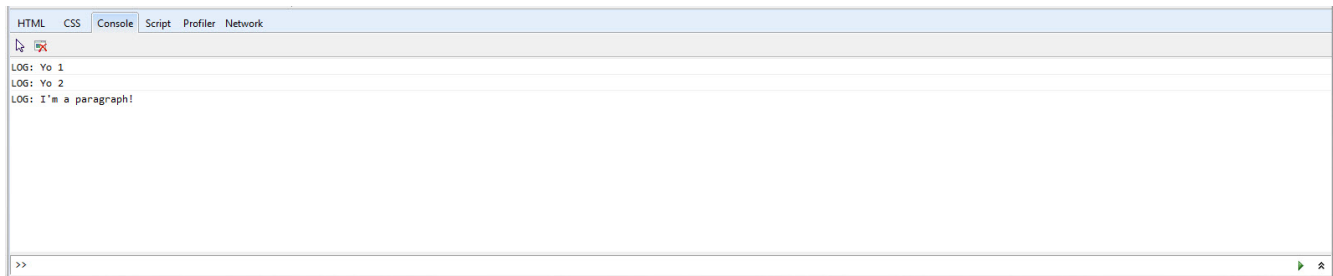
```
<!doctype html>
<html lang="en">
<head>
  <title> The Window Object </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      alert("hey 1");
      window.alert("hey 2");

      console.log("Yo 1");
      window.console.log("Yo 2");

      var p = window.document.getElementById("theP");
      window.console.log(p.innerHTML);
    }
  </script>
</head>
<body>
  <p id="theP">I'm a paragraph!</p>
</body>
</html>
```

Save it in your **javascript 1** folder as **windowobject.html** and click **Preview**. Do you see two alerts? Check the console. Do you see three messages in the console, like this:



You get the same results whether you prefix these functions with **window** or not.

So, why don't you have to type **window.alert("hey");** instead of **alert("hey");** every time? Because window is the **default** object, so JavaScript *assumes* window if you don't type it. That then begs the question, why do we write **window.onload = init;**, when we could just write **onload = init;**? The reason is that while there's usually only one **alert** function, there could be many **onload** properties being set in your code. So it's best to be absolutely clear, and specify **window.onload** so there's no ambiguity about what you mean. This will create fewer bugs and make your code easier to read.

We'll come back to the window object again later; for now, it's just important that you know it's another important object in JavaScript, like document, and that it is the "default object" that has properties and methods you'll use often when writing JavaScript programs.

Another action-packed lesson! In this lesson you've learned:

- The browser represents pages internally using the Document Object Model, which we often visualize as an upside-down tree.
- The Document Object Model (DOM) contains all the elements and content of your page as objects.
- You can access these objects using **document.getElementById()** and **document.getElementsByTagName()**.
- **document.getElementById()** returns one element, the element with the id you specify.
- **document.getElementsByTagName()** returns a collection of elements that match the tag name you specify. Even if only one (or no) elements match, you'll get an array back.
- You can loop through the collection of elements returned by **document.getElementsByTagName()** just like you would loop through an array of items.
- It's important not to access the objects in the DOM until the page has completely loaded.
- The window object is another important object you'll use in your JavaScript programs.
- The window object is the "default object" for your JavaScript programs.

Spend some time practicing with **document.getElementById()** and **document.getElementsByTagName()** before you do the project and go on the next lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Functions and Events

Lesson Objectives

When you complete this course, you will be able to:

- define a function using a function keyword.
- use functions with multiple parameters.
- use different functions together.
- handle the events that you use in JavaScript.
- name your functions appropriately.

What is a Function?


You've already seen a couple of functions in action in this course, but we haven't actually defined a function yet.

A function is a reusable chunk of code. When you put code into a function, you can *call* the function again and again to reuse its code.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>
    function countWords() {
      var sentence = "The answer to life, the universe, and everything is 42";
      var words = sentence.split(" ");
      var numWords = words.length;
      return numWords;
    }

    var howManyWords = countWords();
    alert(howManyWords + " words in the sentence.");
  </script>
</head>
<body>
</body>
</html>
```

Save it in your `/javascript 1` folder as `function.html` and click . You see an alert displaying the number of words in the sentence. How does it work?

OBSERVE:

```
<script>
  function countWords() {
    var sentence = "The answer to life, the universe, and everything is 42";
    var words = sentence.split(" ");
    var numWords = words.length;
    return numWords;
  }

  var howManyWords = countWords();
  alert(howManyWords + " words in the sentence.");
</script>
```

We first *define* our function, using the **function** keyword. We give the function a **name**, which must be followed by **parentheses ()**, and then we define the **body** of the function. We delimit the **body** using the **curly braces {}**. All the

JavaScript statements between the braces are executed every time you call the function, which you do by using its **name followed by ()**.

In this example, the function **countWords()** **returns** a value. That means that the result of calling the function is the value we **return** at the end of the function. When you return from a function, no statements that might follow the **return** statement are executed, so **return** is usually the last statement in the body of the function. You can put the return value of the function into a variable, and use it elsewhere in the program, like we did with the variable **howManyWords**.


The **countWords()** function figures out the number of words in a sentence by using the **split()** function to make an array of all the words, and then finding how many words there are by getting the **length** of the array, and **returns** that value so the code that calls the function gets that value as the result of the function call.

Notice that in JavaScript, we don't have to define a function before we use it:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>
    var howManyWords = countWords();
    alert(howManyWords + " words in the sentence.");

    function countWords() {
      var sentence = "The answer to life, the universe, and everything is 42";
      var words = sentence.split(" ");
      var numWords = words.length;
      return numWords;
    }
    var howManyWords = countWords();
    alert(howManyWords + " words in the sentence.");
  </script>
</head>
<body>
</body>
</html>
```

Save it and click **Preview** . You should see the same alert again. This might seem really odd to you; how can you use a function before it's even defined? JavaScript goes through your code and looks for all function definitions before it starts executing your code, so it knows about a function you've defined below the code that uses it.

Function Parameters and Arguments

Now at this point, you might be saying "Okay, that's great, but what's the point of having a function that does *exactly the same thing* every time I call it? How often is that really going to be useful?"

When you need a function to do something *slightly different* each time you call it, you use *parameters* and *arguments*. Let's take a look at an example, and then we'll talk about the details.


CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>

    function countWords(sentence) {
      var sentence = "The answer to life, the universe, and everything is 42";
      var words = sentence.split(" ");
      var numWords = words.length;
      return numWords;
    }

    var howManyWords = countWords("The answer to life, the universe, and everything is
42");
    alert(howManyWords + " words in the sentence.");

    var howManyWords = countWords("A short sentence");
    alert(howManyWords + " words in the sentence.");
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . You'll see two alerts, the first one with the number of words in the sentence, "The answer to life, the universe, and everything is 42" (10), and the second with the number of words in the sentence "A short sentence" (3). Do you see how a function is useful when you can *customize* it like this?

OBSERVE:

```
<script>
  function countWords(sentence) {
    var words = sentence.split(" ");
    var numWords = words.length;
    return numWords;
  }

  var howManyWords = countWords("The answer to life, the universe, and everything is
42");
  alert(howManyWords + " words in the sentence.");

  var howManyWords = countWords("A short sentence");
  alert(howManyWords + " words in the sentence.");
</script>
```

We added a parameter named **sentence** to the function definition. When you put a variable name inside the parentheses in the function definition like we did here, you're saying "This function expects one value to be passed in." Then, in the body of the function, you can use that variable just like if you had declared and initialized the variable inside the function body. Notice that you do *not* use the **var** keyword for parameter names.

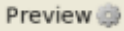
Now, to call a function with a parameter, you have to use an **argument**. An argument is a value that you put between the parentheses when you *call* the function. In this example, we type the value we want to pass to the parameter in between the parentheses in the function call.

Now, let's make another change:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>
    function countWords(sentence) {
      var words = sentence.split(" ");
      var numWords = words.length;
      return numWords;
    }

    var testSentence = prompt("Enter a sentence, and I'll find how many words it has:");
;
    var howManyWords = countWords(testSentence);
    alert(howManyWords + " words in the sentence.");
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . You'll be prompted to enter a sentence and then alerted with the number of words in that sentence. Now you can enter any sentence you want without even having to change the code!

OBSERVE:

```
<script>
  function countWords(sentence) {
    var words = sentence.split(" ");
    var numWords = words.length;
    return numWords;
  }

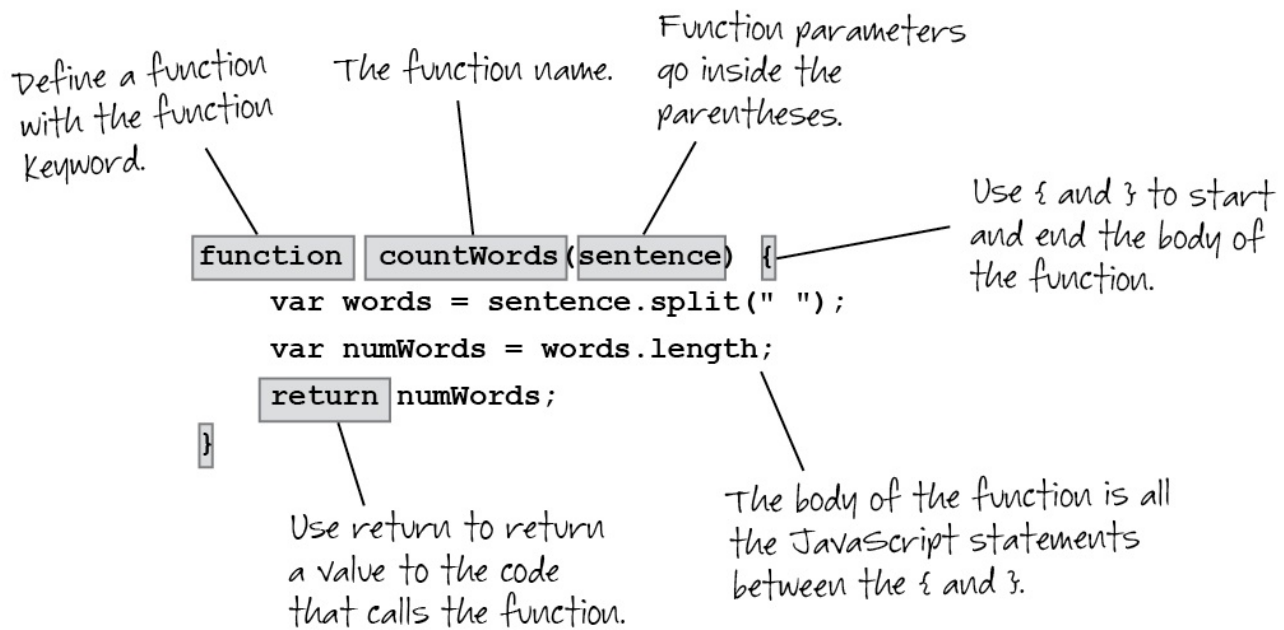
  var testSentence = prompt("Enter a sentence, and I'll find how many words it has:");
;
  var howManyWords = countWords(testSentence);
  alert(howManyWords + " words in the sentence.");
</script>
```

In this case, we used a variable name for the function argument; we initialized the **testSentence** variable to a string with the value of the sentence you typed in, and then used the **testSentence** variable as the argument.

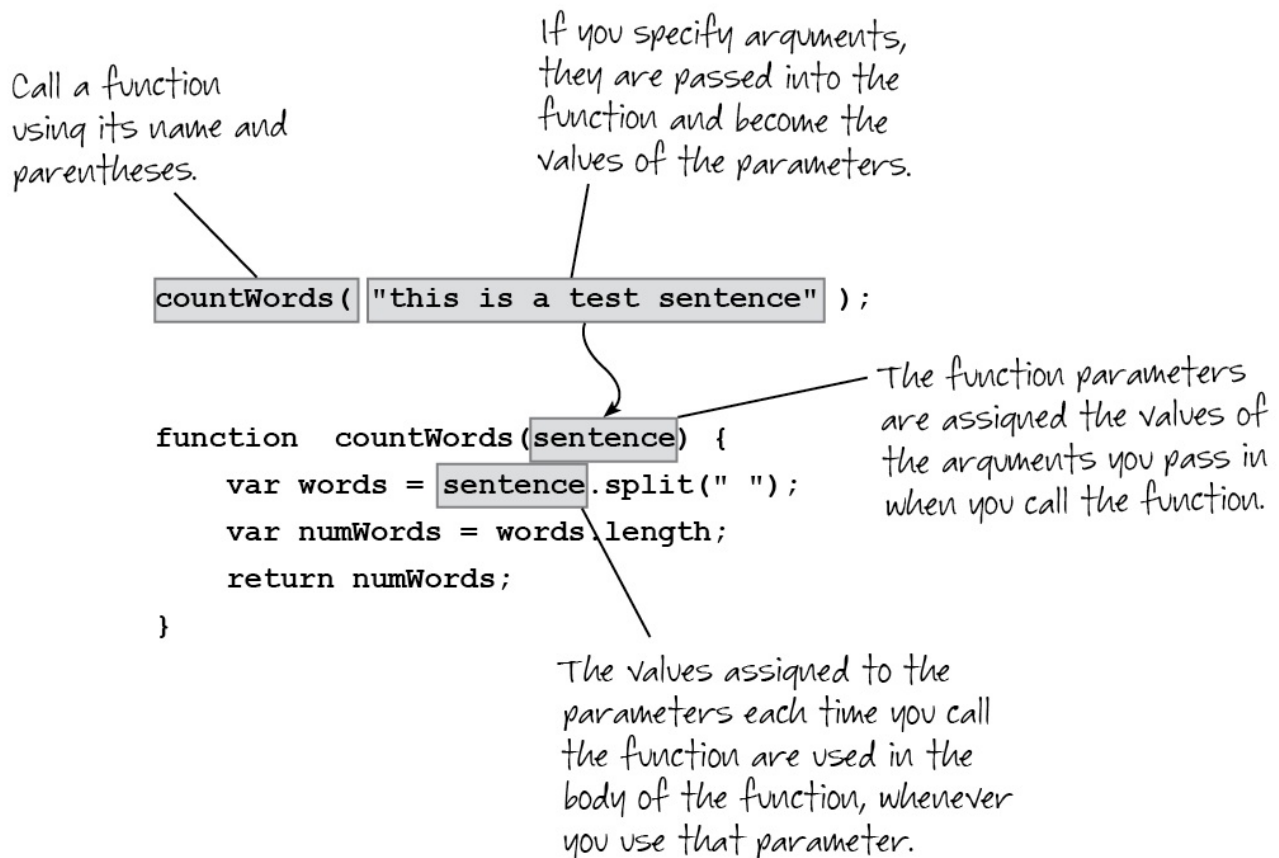
One important thing to notice here is that when you pass an argument to a parameter, you're passing the *value*. Even if you use a variable name as the argument in a function call, what gets stored in the parameter is the *value* of the argument. The parameter is a *different variable*. So in this example, the variable **testSentence** used as the argument is different from the **sentence** variable used as the parameter. It's the *value* that's stored in **testSentence** that gets passed. This is called *passing by value* and it's a common feature of many programming languages.

Also notice that we used different variable names for the argument and the parameter. The argument variable is named **testSentence** and the parameter variable is named **sentence**. That's perfectly fine because it's the *value* that matters. You can give the argument variable and the parameter variable the same name if you want—it's up to you. The code in the body of the function will always use the name of the parameter variable.

Let's review. Here's how you define a function:



And here's what happens when you call a function:




Functions with Multiple Parameters

Functions can take more than one parameter. Just remember that for each parameter, you need to pass in an argument, so the number of arguments matches the number of parameters. Create a new HTML file as shown:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Functions with Multiple Parameters </title>
  <meta charset="utf-8">
  <script>
    function areaRect(width, height) {
      var area = width * height;
      return area;
    }

    var area = areaRect(3, 2);
    alert(area);
  </script>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript 1** folder as **functionmulti.html** and click [Preview](#) . In this example, the argument **3** is passed into the parameter **width**, and the argument **2** is passed into the parameter **height**. Arguments and parameters are matched in order, so the first argument will always be the value of the first parameter, and so on.

Multiple Functions

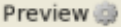
You can use functions together, and in fact, you'll find that you typically have many functions in your JavaScript. Each task that you want to do potentially more than once is a good candidate to put into a function. (You'll also find that you'll reuse functions that do common tasks in different web applications!). Reopen your **function.html** and edit it as shown below:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Multiple Functions </title>
  <meta charset="utf-8">
  <script>
    function getSentence() {
      var sentence = prompt("Please enter a sentence");
      if (sentence == null || sentence == "") {
        alert("Please enter some words!");
      }
      else {
        var howManyWords = countWords(sentence);
        alert(howManyWords + " words in the sentence.");
      }
    }

    function countWords(sentence) {
      var words = sentence.split(" ");
      var numWords = words.length;
      return numWords;
    }

    getSentence();
var testSentence = prompt("Enter a sentence, and I'll find how many words it has:");
var howManyWords = countWords(testSentence);
alert(howManyWords + " words in the sentence.");
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . You'll be prompted to enter a sentence, and you'll see an alert with the number of words in that sentence.

OBSERVE:

```
<script>
  function getSentence() {
    var sentence = prompt("Please enter a sentence");
    if (sentence == null || sentence == "") {
      alert("Please enter some words!");
    }
    else {
      var howManyWords = countWords(sentence);
      alert(howManyWords + " words in the sentence.");
    }
  }

  function countWords(sentence) {
    var words = sentence.split(" ");
    var numWords = words.length;
    return numWords;
  }

  getSentence();
</script>
```

Here, we have two functions: **getSentence()**, which prompts the user and makes sure they typed something; and **countWords()**, which counts the words of the sentence passed to it. This splits up the work and makes the code easier to reuse.

We **call** `countWords()` from the `getSentence()` function. And, of course, to start everything going, we need to **call** `getSentence()` as well.

Handling Events

You'll need functions whenever you use *events* in JavaScript, which will be in almost every program you write. An *event* is something that happens in your JavaScript program that you can choose to *handle*. Events can be generated by you (like if you click a mouse), or generated by the browser (like if the page has completed loading), or even generated by JavaScript itself (like if a timer goes off).

We're going to come back to events in more detail later in the course, but for now, the important thing to note is that you'll need a function whenever you want to handle an event. These functions are often called—strangely enough—*event handlers*. Let's update our `function.html` program to use a form. When you use the form to submit a sentence, your program will need **handle** to get the data from the form to process.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Multiple Functions </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var button = document.getElementById("submit");
      button.onclick = getSentence;
    }


    function getSentence() {
      var sentence = prompt("Please enter a sentence");
      var sentenceInput = document.getElementById("sentence");
      var sentence = sentenceInput.value;
      if (sentence == null || sentence == "") {
        alert("Please enter some words!");
      }
      else {
        var howManyWords = countWords(sentence);
        alert(howManyWords + " words in the sentence.");
      }
    }

    function countWords(sentence) {
      var words = sentence.split(" ");
      var numWords = words.length;
      return numWords;
    }

    getSentence();
  </script>
</head>
<body>

  <form>
    <label for="sentence">Enter a sentence: </label>
    <input type="text" id="sentence" size="20" value=""> <br>
    <input type="button" id="submit" value="Get the number of words!">
  </form>

</body>
</html>
```

Save it and click . Type in a sentence to test. When you click the "Get the number of words!" button, you should see an alert showing the number of words in the sentence.

Functions as Event Handlers

OBSERVE:

```
<script>
  window.onload = init;

  function init() {
    var button = document.getElementById("submit");
    button.onclick = getSentence;
  }

  function getSentence() {
    var sentenceInput = document.getElementById("sentence");
    var sentence = sentenceInput.value;
    if (sentence == null || sentence == "") {
      alert("Please enter some words!");
    }
    else {
      var howManyWords = countWords(sentence);
      alert(howManyWords + " words in the sentence.");
    }
  }

  function countWords(sentence) {
    var words = sentence.split(" ");
    var numWords = words.length;
    return numWords;
  }
</script>
```

This has three functions. The first function, `init()`, runs when the page finishes loading. We say "`init()` is the `load` event handler." Now that you know a bit more about functions, take another look at how we assign a function to the `window.onload` property. We write:

```
window.onload = init;
```

We *don't* write:

```
window.onload = init();
```

Why? You know the answer to this now, right? Because when we write `init()`, we're *calling* the function `init`, which we don't want to do. We want JavaScript to call it *for* us, after the `load` event has been triggered, which the browser does for us when it finishes loading the page. We use a **load event handler** so we can access the DOM safely—that is, only *after* the page has finished loading. If we typed `window.onload = init();`, then JavaScript would try to run the `init()` function as soon as it saw that line of code, which is too soon. By assigning `window.onload` to the function *name*, we let JavaScript run the function when the page is ready.

In `init()`, we set up a button click handler, so the `getSentence()` function is called when you click the button. And here, we say "`getSentence()` is the `click` handler for the button." This works exactly like `window.onload` does; we want JavaScript to call the `getSentence()` function only when we click the button, which generates a **click event**, so we use the function name when we assign the **click event handler** to the button.

Setting up handlers like this is something you'll do often in your JavaScript code. Many events, like **click** and **load**, have corresponding properties, like **onclick** and **onload**, that you can use to set them up.

Walking Through the Rest of the Code

The function `getSentence()` gets the value of the "sentence" element (that is, the form input with the id "sentence"). This holds the value you typed in for the sentence. It does this first by getting the element itself, and then by getting its value. (You'll learn much more about this in the upcoming Forms lesson!).

Once we have the value of the sentence, just like before we test to see if the value is not empty by comparing to null, and comparing to the empty string, "". If the sentence is empty, then we alert the user to ask her to enter some words. If the sentence is not empty, then we call the `countWords()` function, passing in the string as the argument.

Finally, the **countWords()** function is the same as it was before; it computes the number of words in a string of words that's passed into the parameter `sentence` and returns that value.

One last little thing to notice: we don't have to call **getSentence()** ourselves anymore. Why? Because the function is being called when we click the form button! **getSentence()** is our click handler, so it's called when we click the button. Think through again how this works.

When you have multiple functions like this, it can get tricky to follow what's often called the *flow of execution*, because you have functions calling other functions, which are returning results, and events can happen which cause other functions to be called. Sometimes it helps to print out the code and draw on the paper, with lines showing which function is calling which other function and when. Number the lines so you know the order in which things happen. Try doing this for the example above.

Naming Functions

Before we end this lab, we should talk about naming functions. Notice that we use descriptive names, just like we do for variables. In general, the rules for function names are the same as those for variables. So stick with those and you'll be fine. Make sure you use camelCase if you're using multi-word names, like we did in this lab.

Also, notice that when we're describing a function and we say something like, "Then call function **countWords()** to figure out how many words there are in the sentence," we write **countWords()** rather than **countWords**. This is often done in books and other descriptions of functions so that you can easily distinguish that we're talking about a function rather than just a variable. But you may sometimes see them written without the () as well. Either way is fine when you're *describing* your code in written text, but when you're actually *writing* code, it's really important to know when to use the parentheses and when not to, right?!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Scope

Lesson Objectives

When you complete this course, you will be able to:

- use variables and demonstrate how they are impacted by local and global scope.
- use local and global variables.

Variable Levels

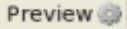
You've used variables in three different ways so far in this course. You've used variables at the *top level*, by adding them to a `<script>` element like this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>

    var circleRadius = 3;
    console.log("Radius: " + circleRadius); // Use alert instead of console.log if you want.

  </script>
</head>
<body>
</body>
</html>
```

Save it in your `/javascript 1` folder as `scope.html` and click . In the console, you should see the value 3 for the radius.


You've also used variables within **functions**, like this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>
    var circleRadius = 3;
    console.log("Radius: " + circleRadius);

    var area = circleArea();
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea() {
      var circleRadius = 5;
      console.log("Radius in the function circleArea(): " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }
  </script>
</head>
<body>
</body>
</html>
```

We used the **circleRadius** variable only inside the **circleArea()** function. Save it and click . What values do you see in the console? Do you know why you are seeing these values?

Finally, we've also used variables as *parameters* of functions:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>
    var circleRadius = 3;
    console.log("Radius: " + circleRadius);

    var area = circleArea(5);
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea(circleRadius) {
      var circleRadius = 5;
      console.log("Radius in the function circleArea(): " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }
  </script>
</head>
<body>
</body>
</html>
```

In this case, we pass the argument **5** into a parameter, **circleRadius**. Now what value do you see in the console when you save and click Preview?

So, what's the difference?

Global Scope and Local Scope

In most computer languages, there is the concept of **scope**. Scope is the visibility of a variable—that is, which parts of your code can "see" the value of a given variable.

JavaScript has two kinds of scope: *global* and *local*. Let's tackle global scope first.

Global scope is at the top level of your JavaScript code. It's code that's *not* in a function:

OBSERVE:

```
<script>

  var circleRadius = 3;
  console.log("Radius: " + circleRadius);

</script>
```


Here, the variable **circleRadius** is a *global* variable. So what? Well, a global variable has *global scope*, which means it's visible *everywhere*! Take a look:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>
    var circleRadius = 3; // this is a global variable
    console.log("Radius before we call circleArea(): " + circleRadius);

    var area = circleArea(5);
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea(circleRadius) {
      console.log("Radius in the function circleArea(): " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }
  </script>
</head>
<body>
</body>
</html>
```

Make sure you type the code above from scratch or edit it correctly. This version should have no function parameter for **circleArea()**. Save it and click **Preview** . These messages display in the console:

OBSERVE:

```
Radius before we call circleArea(): 3
Radius in the function circleArea(): 3
Radius after we call circleArea(): 3
```

OBSERVE:

```
<script>
  var circleRadius = 3;
  console.log("Radius before we call circleArea(): " + circleRadius);

  var area = circleArea();
  console.log("Radius after we call circleArea(): " + circleRadius);

  function circleArea() {
    console.log("Radius in the function circleArea(): " + circleRadius);
    var area = Math.PI * Math.pow(circleRadius, 2);
    return area;
  }
</script>
```

Notice that you didn't get an error for the **console.log message** in the function **circleArea()**. That means that the value of the **circleRadius** variable is **visible inside the circleArea()** function even though it's defined *outside* the function, in the global scope. You're using the *same* variable in all three console messages.


What do you think will happen if you change the value of **circleRadius** inside the function? Try setting **circleRadius** to another value inside the function and add another console.log or alert message to display the value of **circleRadius** after you call the function, like this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>
    var circleRadius = 3;
    console.log("Radius before we call circleArea(): " + circleRadius);

    var area = circleArea();
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea() {
      console.log("Radius in the function circleArea(): " + circleRadius);
      circleRadius = 5;
      console.log("Radius in the function circleArea(), changed: " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius, 2);
      return area;
    }
  </script>
</head>
<body>
</body>
</html>
```

Save it and click . The following displays in the console:

OBSERVE:

```
Radius before we call circleArea(): 3
Radius in the function circleArea(): 3
Radius in the function circleArea(), changed: 5
Radius after we call circleArea(): 5
```

Why?

OBSERVE:

```
<script>
  var circleRadius = 3;
  console.log("Radius before we call circleArea(): " + circleRadius);

  var area = circleArea();
  console.log("Radius after we call circleArea(): " + circleRadius);

  function circleArea() {
    console.log("Radius in the function circleArea(): " + circleRadius);
    circleRadius = 5;
    console.log("Radius in the function circleArea(), changed: " + circleRadius);
    var area = Math.PI * Math.pow(circleRadius,2);
    return area;
  }
</script>
```

You might have expected that last console.log message to display the value 3, which is the initial value of **circleRadius**. But in the **circleArea()** function, you *change* the value of the **circleRadius** variable to 5. The variable **circleRadius** is *declared* in the global scope, that is, outside the **circleArea()** function. So you can see its value in the function **circleArea()** and you can *change* its value in that function too. So, when the function completes, the new value of the global variable, **circleRadius** is now 5.

Here's a picture of the global scope of the variable **circleRadius**:

We're declaring the variable **circleRadius** here, at the top-level, outside of any function. It's a "global variable."

The scope of the **circleRadius** variable is global. That means you can see and change the value everywhere.

```
var circleRadius = 3;
console.log("Radius before we call circleArea(): " + circleRadius);

var area = circleArea();
console.log("Radius after we call circleArea(): " + circleRadius);

function circleArea() {
  console.log("Radius in the function circleArea(): " + circleRadius);
  circleRadius = 5;
  console.log("Radius in the function circleArea(), changed: " + circleRadius);
  var area = Math.PI * 2 * circleRadius;
  return area;
}
```

If you change the value of the global variable **circleRadius** in the function **circleArea()**, you are changing the value everywhere.

Local Scope

Local scope means that the visibility of variables is limited. Let's change the example above a bit and move the


declaration of the **circleRadius** variable from outside the **circleArea()** function to inside the function and see what happens. Can you guess what's going to happen?

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>
    var circleRadius = 3;
    console.log("Radius before we call circleArea(): " + circleRadius);

    var area = circleArea();
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea() {
      console.log("Radius in the function circleArea(): " + circleRadius);
      var circleRadius = 5;
      console.log("Radius in the function circleArea(), changed: " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }
  </script>
</head>
<body>
</body>
</html>
```

Make sure you add the word **var** in front of the variable **circleRadius** in the function **circleArea()**! This is important. Save it and click **Preview** . In the console, you see one message that shows the "Radius in the function circleArea(): 5" (that is, the value of the **circleRadius** variable inside the function is 5), and then you should see an error that says something like, "ReferenceError: Can't find variable: circleRadius." You don't see the console message for the "Radius after we call circleArea()" console.log.

Why do you think this is?

The **circleArea()** function creates a local scope that includes only the body of the function. That means if you declare any variables inside the function, they are *not visible* outside the function!

Here's a picture of the local scope of **circleRadius**:

Now the scope of `circleRadius` is local to `circleArea()`. That means you can see and change the value only in the `circleArea()` function.

You can't see the value of `circleRadius` here, because `circleRadius` is only visible in its local scope, the function `circleArea()`.

```
var area = circleArea();  
console.log("Radius after we call circleArea(): " + circleRadius);  
  
function circleArea() {  
  var circleRadius = 5;  
  console.log("Radius in the function circleArea(): " + circleRadius);  
  var area = Math.PI * 2 * circleRadius;  
  return area;  
}
```

Notice that we're declaring the variable `circleRadius` in the function `circleArea()`, using the keyword `var`. It's a "local variable."

The reason you get an error message for the `console.log` message that tries to display the value of **`circleRadius`** *after* the call to **`circleArea()`** is because **`circleRadius`** is not defined in the global scope, so this `console.log` message can't see its value. It's "hidden" inside the local scope of the function **`circleArea()`**.

Local scope is also different from global scope in that the variables defined in a function last only as long as the call to the function. So if you call a function once, the variables exist for a short while and then they go away as soon as the function ends. If you call the function again, *new versions* of those variables exist, again for a short period, until the function ends. Global variables, however, last as long as your page is open!

Shadowing

Now, what happens if you have a global **`circleRadius`** variable *and* a local **`circleRadius`** variable? Let's see.


CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>
    var circleRadius = 3;
    console.log("Radius before we call circleArea(): " + circleRadius);

    var area = circleArea();
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea() {
      var circleRadius = 5;
      console.log("Radius in the function circleArea(): " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }

  </script>
</head>
<body>
</body>
</html>
```

Save it and click . What do you see in the console?

OBSERVE:

```
<script>
  var circleRadius = 3;
  console.log("Radius before we call circleArea(): " + circleRadius);

  var area = circleArea();
  console.log("Radius after we call circleArea(): " + circleRadius);

  function circleArea() {
    var circleRadius = 5;
    console.log("Radius in the function circleArea(): " + circleRadius);
    var area = Math.PI * Math.pow(circleRadius,2);
    return area;
  }

</script>
```

We declared *two* **circleRadius** variables. The **one outside the function** has the value 3, and the **one inside the function** has the value 5.

The **console.log** that displays the value of the **global variable** before we call **circleArea()** displays 3, while the **console.log** that displays the value of the **local variable** displays 5. And finally, the **console.log** that displays the value of the **global variable** after the call to **circleArea()** displays 3.

In the console, you see:

OBSERVE:

```
Radius before we call circleArea(): 3
Radius in the function circleArea(): 5
Radius after we call circleArea(): 3
```

So what's going on here? First, because we *declared* **circleRadius** twice, in two *different* scopes, we created two completely different variables! Second, the local variable *shadows* the global variable. That means that inside the function, if you declare a variable with the *same name* as a global variable, you're creating a separate variable that is used within the function instead of the global variable. So when we **display the value of circleRadius** in the function

`circleArea()`, and we see 5, we are seeing the value of the **local variable**, which we've declared to be 5. When we display the value of the **circleRadius** variable **after the call to `circleArea()`**, we are seeing the value of the **global variable**, which is 3.

Here's a picture of shadowing:

This variable has global scope.

```
var circleRadius = 3;
console.log("Radius before we call circleArea(): " + circleRadius);

var area = circleArea();
console.log("Radius after we call circleArea(): " + circleRadius);

function circleArea() {
  var circleRadius = 5;
  console.log("Radius in the function circleArea(): " + circleRadius);
  var area = Math.PI * 2 * circleRadius;
  return area;
}
```

This variable has local scope.

Notice that the local scope "shadows" the global scope. So inside `circleArea()`, the only value of `circleRadius` that is visible is the local variable.

To really see the difference between local and global scope, make one small change:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>


    var circleRadius = 3;
    console.log("Radius before we call circleArea(): " + circleRadius);

    var area = circleArea();
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea() {
      var circleRadius = 5;
      console.log("Radius in the function circleArea(): " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }

  </script>
</head>
<body>
</body>
</html>
```


What do you think the **console.log** outside the function will display now?

Save it and click . You see these console messages:

OBSERVE:

```
Radius before we call circleArea(): 3
Radius in the function circleArea(): 5
Radius after we call circleArea(): 5
```

Why? Because now the **circleRadius** variable that we're setting to 5 inside the function is *the same* global variable that we declared outside the function. So when we change the value to 5 inside the function, it changes the value of that global variable, so when we **display the value of that variable** after the call to **circleArea()**, we see that new value.

This is a bit tricky, so step through it all again and make sure you understand the difference between a global variable and a local variable, and when a local variable will shadow a global variable.

Scope of Function Parameters

There's one more use case we need to look at to completely understand scope. What is the scope of a function parameter?

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>
    var circleRadius = 3;
    console.log("Radius before we call circleArea(): " + circleRadius);

    var area = circleArea(circleRadius);
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea(circleRadius) {
      circleRadius = 5;
      console.log("Radius in the function circleArea(): " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }

  </script>
</head>
<body>
</body>
</html>
```

Here, we removed the declaration of **circleRadius** inside the function, and we passed the value of the global variable **circleRadius** into the function parameter.

As you might expect, the value of the *parameter*, **circleRadius**, inside the function is 3, because that's the value of the variable we pass as the argument when we **call the circleArea() function**. So the console messages display the same value.

But what happens if we *change* the value of **circleRadius** inside the circleArea() function?


CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Scope </title>
  <meta charset="utf-8">
  <script>

    var circleRadius = 3;
    console.log("Radius before we call circleArea(): " + circleRadius);

    var area = circleArea(circleRadius);
    console.log("Radius after we call circleArea(): " + circleRadius);

    function circleArea(circleRadius) {
      circleRadius = 5;
      console.log("Radius in the function circleArea(): " + circleRadius);
      var area = Math.PI * Math.pow(circleRadius,2);
      return area;
    }
  </script>
</head>
<body>
</body>
</html>
```

Save it and click  , and check the console. What happened?

OBSERVE:

```
<script>

  var circleRadius = 3;
  console.log("Radius before we call circleArea(): " + circleRadius);

  var area = circleArea(circleRadius);
  console.log("Radius after we call circleArea(): " + circleRadius);

  function circleArea(circleRadius) {
    circleRadius = 5;
    console.log("Radius in the function circleArea(): " + circleRadius);
    var area = Math.PI * Math.pow(circleRadius,2);
    return area;
  }
</script>
```

The value of **circleRadius** displayed by the console message **inside the function is 5**, and the value displayed by the console message **outside the function** and **after** we've called **circleArea()** is 3.

That means that the parameter **circleRadius** is a *local variable*, just as if we'd declared it inside the function. It's a completely different variable from the other, global variable **circleRadius**, so when we change the **circleRadius** variable *inside* the function, it does *not* affect the global **circleRadius** variable *outside* the function.

In other words, function parameters are local variables with local scope.

Remember in the last lesson when we said that JavaScript is "pass by value"? When you call a function, you're passing the *value* of the argument into the parameter, which is a separate variable. And now you know that the parameter is a separate variable, with local scope.

Different Functions have Different Scope

This might be obvious to you at this point, but it's worth noting: two different functions that use the same local variable name have two different scopes, so the variables don't interfere with each other. Take a look at this example.

OBSERVE:

```
<script>
  console.log("Circle area: " + circleArea(3));
  console.log("Square area: " + squareArea(6));

  function circleArea(circleRadius) {
    var area = Math.PI * Math.pow(circleRadius,2);
    return area;
  }

  function squareArea(length) {
    var area = length * length;
    return area;
  }

</script>
```

Now, we've got two different functions, both of which declare a local variable **area**. These two variables are completely different variables, so setting one won't affect the other. The scope of each of these variables is limited to the function it is in.

Using Global and Local Variables

Because it's so easy to unintentionally confuse global variables and local variables, and to shadow global variables with local variables, you should try to keep the number of global variables you use to a minimum. When you're writing small programs like we're doing here, it doesn't matter as much. But when you get into writing much bigger programs, and using JavaScript libraries, it's especially important to minimize the use of global variables. You'll see one trick for doing this in the next lab, when we start using objects.

Usually, you'll only need global variables when you have variables that need to be accessed by more than one function, and you can't easily pass a variable from one function to another. You'll also need a global variable if you need to keep a value, say, a counter, that is separate from any functions that are called. Why? Because, when you call a function, the local variables are *created* and *destroyed* each time you call it! So if you want a variable to have a value that lasts beyond a function call, you'll need to use a global variable.

When using local variables, be very careful to always *declare* your local variables in the function where you're using them! For instance, you'd write:

OBSERVE:

```
function squareArea(length) {
  var area = length * length;
  return area;
}
```

rather than:

OBSERVE:

```
function squareArea(length) {
  area = length * length;
  return area;
}
```

Why? Because if you do *not* declare a variable that you intend to be local inside the function, JavaScript will create the variable as a *global* variable, and this could have unintended consequences elsewhere in your program. In the latter example above, it's as if you'd typed:

OBSERVE:

```
var area;  
function squareArea(length) {  
    area = length * length;  
    return area;  
}
```

Try writing your own program using both global and local variables. Experiment. Make sure you understand why you get the behavior you do before you go on.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Objects

Lesson Objectives

When you complete this course, you will be able to:

- create your own objects.
- store objects within an array.
- use arrays as object property values.
- use the constructor function to create objects.
- change the value of a property in an object after by assigning it a new value.
- use objects to collect global variables.

We've mentioned *objects* a few times in this course so far; for instance, you know that the browser represents the web page as a collection of objects in a tree structure that we refer to as the Document Object Model (or DOM). You also know that elements are objects. You might have even picked up that objects have *properties*. Now it's time to look more deeply at what an object actually is, and even create some of your own!

Objects are Collections of Properties

An object is a *collection of properties*. Let's take a look at an example.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    var tilla = {
      type: "dog",
      name: "Tilla",
      weight: 26
    };
  </script>
</head>
<body>
</body>
</html>
```

Save it in your `/javascript 1` folder as `object.html` and click [Preview](#). You won't see anything (because there's nothing in the page). Open the developer console (in case you forgot how to do this, see the [Developer Tools for JavaScript Programming](#)). At the JavaScript console prompt, type `tilla`. You should see something like this:

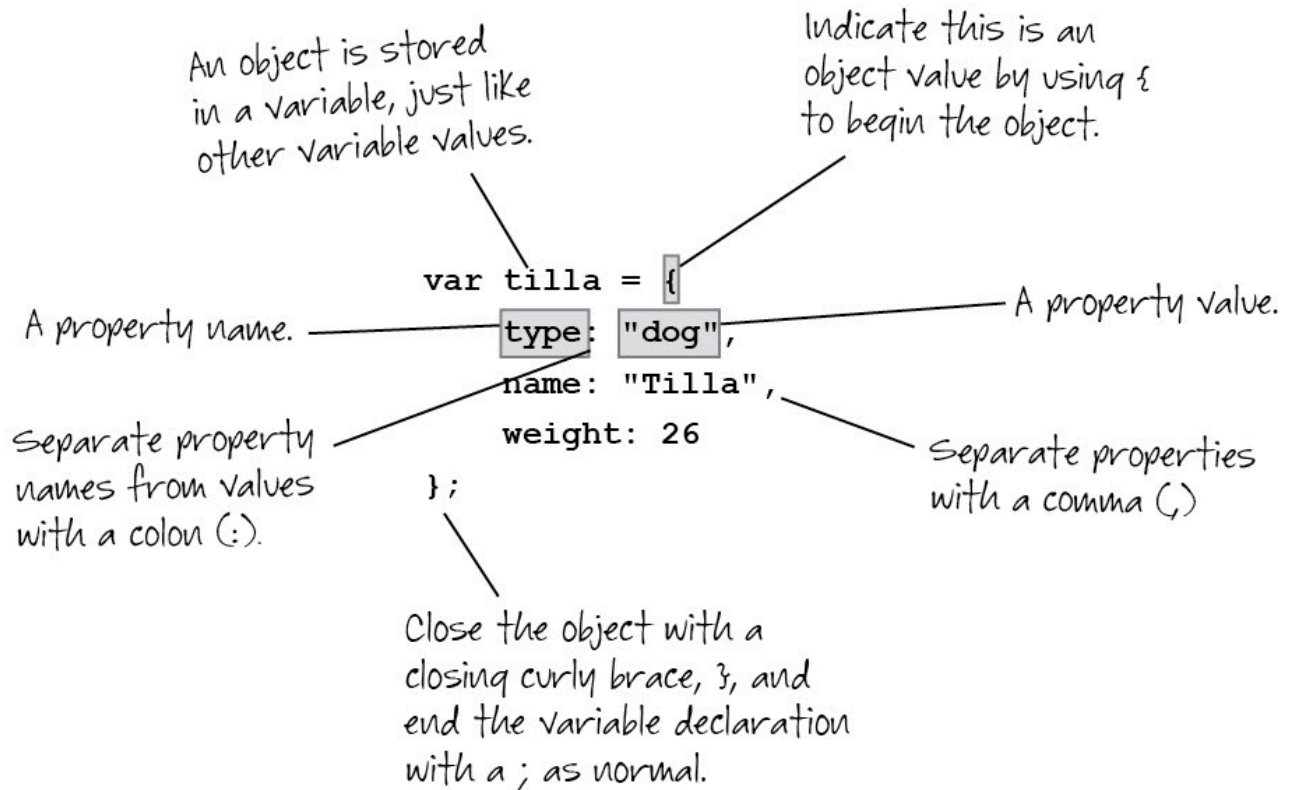


```
> tilla
▼ Object
  name: "Tilla"
  type: "dog"
  weight: 26
  __proto__: Object
>
```

Let's take a closer look at what you just did. Notice that you declare an object just like you would a variable, by starting with the `var` keyword and giving your object a variable name, in this case, `tilla`. You initialize your object variable to an object value. You know it's an object because of the curly braces `{}`. Inside the braces are a collection of *properties*. You can think of each property as being similar to a variable declaration, except, of course, the syntax is different. The **property name** is on the left, then a colon, and then the **property value**. So in this case, you added the property name `type` to the object `tilla`, and this property has the string value "dog." Similarly, you added **name** and **weight** properties. Notice that the **type** and **name** properties have string values, while the **weight** property has an integer

number value. Also notice that you separate each property name/value pair with a comma, except for the last one. Finally, notice that we end the object declaration and initialization with a semicolon, as usual when we're declaring a variable.

Here's a review :



Let's add another object:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    var tilla = {
      type: "dog",
      name: "Tilla",
      weight: 26
    };

    var pickles = {
      type: "cat",
      name: "Pickles",
      weight: 7
    };
  </script>
</head>
<body>
</body>
</html>
```

This object is similar to **tilla**; it's got the **type**, **name**, and **weight** properties, except of course the values of the properties are different, and the variable name is **pickles** instead of **tilla**. Try viewing **pickles** in the console like we did earlier with **tilla**.

Accessing Object Properties

So, now you've got a couple of objects; what can you do with them?

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>


    window.onload = init;

    function init() {
      var tilla = {
        type: "dog",
        name: "Tilla",
        weight: 26
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7
      };

      var div = document.getElementById("pets");
      div.innerHTML =
        "My " + tilla.type + " is named " + tilla.name +
        " and she weighs " + tilla.weight + " pounds.";
    }

  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click . You should now see your web page update with content that shows the text "My dog is named Tilla and she weighs 26 pounds." Notice that we added a `<div>` element to the body of the page, and we're updating that element with values from the `tilla` object.

To access a property in an object, you use *dot notation*. For instance, to access the **type** property of the **tilla** object, you write **tilla.type**.

That is, you write the variable name of the object, and then a dot, and then the property name. The result of this expression is the value of the property, in this case "dog."

You access the **name** and **weight** properties in the same way, using **tilla.name** and **tilla.weight**, which have the values "Tilla" and 26, respectively.

In this example, we used the object values to create a string and then updated the page by setting the content of the "pets" **<div>** to that string.

Now, try writing a string using the **pickles** object instead.

Objects and Arrays

Storing Objects in an Array

You can store objects in an array, just like you can other values:


CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var tilla = {
        type: "dog",
        name: "Tilla",
        weight: 26
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7
      };

      var div = document.getElementById("pets");
      div.innerHTML =
      "My " + tilla.type + " is named " + tilla.name +
      " and she weighs " + tilla.weight + " pounds.";
      var pets = [tilla, pickles];
      for (var i = 0; i < pets.length; i++) {
        var pet = pets[i];
        if (pet.type == "dog") {
          div.innerHTML += pet.name + " says Woof! <br>";
        }
        else if (pet.type == "cat") {
          div.innerHTML += pet.name + " says Meow! <br>";
        }
      }
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click . You should see a page that looks like this:

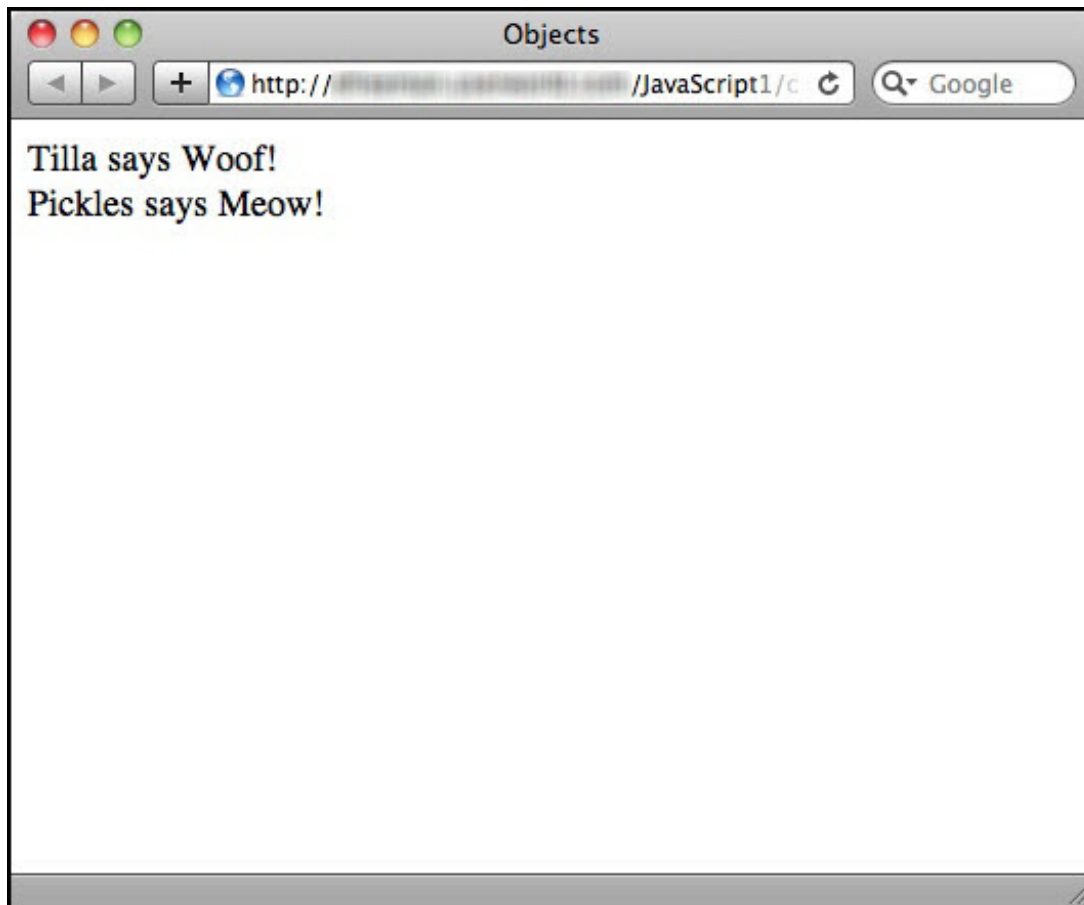
```
Tilla says Woof!
Pickles says Meow!
```


OBSERVE:

```
var div = document.getElementById("pets");

var pets = [tilla, pickles];
for (var i = 0; i < pets.length; i++) {
    var pet = pets[i];
    if (pet.type == "dog") {
        div.innerHTML += pet.name + " says Woof! <br>";
    }
    else if (pet.type == "cat") {
        div.innerHTML += pet.name + " says Meow! <br>";
    }
}
```

In this example, we **create an array named `pets`**, of length 2, containing the two objects. You can iterate over the array just like you normally would, **getting each item from the array using its index**. In this case, each time through the loop, the variable **`pet`** is set to the object at index **`i`** in the array. Once you have this variable that contains the object, you can access the object's properties and values, like you normally would; for instance, **`pet.type`** will be "dog" for the first object in the array and "cat" for the second object in the array.



Arrays as Object Property Values

You can also put arrays in objects! Arrays are just like other values that you can put in objects, so let's add a **`likes`** array to each of our pets. (Don't forget to add the comma after the **`weight`** property value, because that's no longer the last property in the object, and you'll get an error if you forget it.)

CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var tilla = {
        type: "dog",
        name: "Tilla",
        weight: 26,
        likes: ["playing ball", "going for walks", "eating anything"]
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7,
        likes: ["sleeping", "eating butter"]
      };

      var div = document.getElementById("pets");
      var pets = [tilla, pickles];
      for (var i = 0; i < pets.length; i++) {
        var pet = pets[i];
        if (pet.type == "dog") {
          div.innerHTML += pet.name + " says Woof! <br>";
        }
        else if (pet.type == "cat") {
          div.innerHTML += pet.name + " says Meow! <br>";
        }
      }

      div.innerHTML = tilla.name + " enjoys ";
      for (var i = 0; i < tilla.likes.length; i++) {
        div.innerHTML += tilla.likes[i];
        if (i < tilla.likes.length - 1) {
          div.innerHTML += " and ";
        }
      }
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click . You should see the page updated with the content, "Tilla enjoys playing ball and going for walks and eating anything". The code you added at the bottom of this program loops through all the items in the **tilla.likes** array. Just like you normally would with an array, you can use the **length** property to get the length of your array and use it in the for loop. Notice that we accessing each item in the array with **tilla.likes[i]**, using the same syntax you normally would for an array, only using the dot notation to access the array. We also added a check to add an " and " to the string to separate each likes item for display. We check to see if we're not on the last item, and if we're not, we add the " and ".

Go ahead and write the code to do the same thing for the **pickles** object and make sure you get the results you expect.

CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

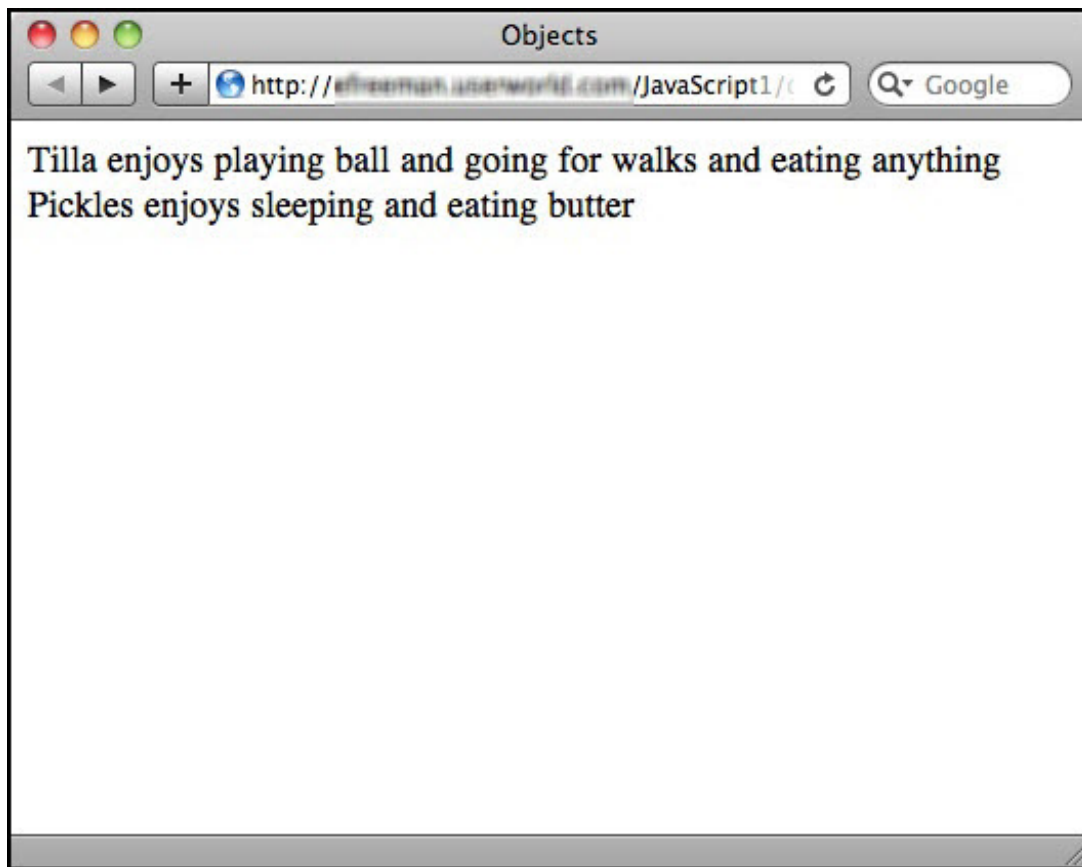
    function init() {
      var tilla = {
        type: "dog",
        name: "Tilla",
        weight: 26,
        likes: ["playing ball", "going for walks", "eating anything"]
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7,
        likes: ["sleeping", "eating butter"]
      };

      var div = document.getElementById("pets");

      div.innerHTML = tilla.name + " enjoys ";
      for (var i = 0; i < tilla.likes.length; i++) {
        div.innerHTML += tilla.likes[i];
        if (i < tilla.likes.length - 1) {
          div.innerHTML += " and ";
        }
      }
      div.innerHTML += "<br>" + pickles.name + " enjoys ";
      for (var i = 0; i < pickles.likes.length; i++) {
        div.innerHTML += pickles.likes[i];
        if (i < pickles.likes.length - 1) {
          div.innerHTML += " and ";
        }
      }
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click [Preview](#) . Do you see Pickle's likes in your web page? You should see this:



How are Objects and Arrays Similar and Different?

You might be thinking that objects have some similarities with arrays, and you'd be right. They both "collect" things together. However, as you can see, objects collect properties, which are pairs of names and values, while arrays collect just values. And you access an object property using the property name, while you access array items using an index.

Object Constructors

Now that you know how to create objects, what do you think you'd do if we asked you to create 100 different pets? Or even just 10? You might think that it would get awfully tedious to keep creating the same kind of object over and over and over... but don't worry, we won't ask you to create any more pet objects like this, because there is a better way: you can use an *object constructor* instead.

You probably noticed that **tilla** and **pickles** have a lot in common, right? They both have a type, a name, a weight, and an array of likes. The *values* of these properties are different, but they are the same properties.

So, let's create a special kind of function, called a **constructor**, that we can use to **construct objects**. For this example, go ahead and create a new file (saving your previous work separately).


CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Object Constructors </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function Pet(type, name, weight, likes) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting", "eating"]);

      var div = document.getElementById("pets");
      div.innerHTML = annie.name + " is a " + annie.type + " and " + willie.name +
        " is a " + willie.type;
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it in your `/javascript1` folder as `constructor.html` and click [Preview](#) . The "pets" <div> is updated with the content "Annie is a cat and Willie is a dog."

OBSERVE:

```
<script>
  window.onload = init;

  function Pet(type, name, weight, likes) {
    this.type = type;
    this.name = name;
    this.weight = weight;
    this.likes = likes;
  }

  function init() {
    var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
    var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting", "eating"]);

    var div = document.getElementById("pets");
    div.innerHTML = annie.name + " is a " + annie.type + " and " + willie.name +
      " is a " + willie.type;
  }
</script>
```

Now, instead of creating pets by writing out the object each time (also known as an *object literal*), we call a function **Pet**. Notice that we used an uppercase "P" to start the name of the function; this is a convention JavaScript programmers use to distinguish *constructor functions* from regular functions.

The **Pet** constructor function creates a pet object with each of the properties of a pet: type, name, weight, and likes. So to create a pet object, we need to call the **Pet** constructor function and pass in all the property values we want for that

pet. But notice something else: when we called the **Pet** constructor, we called it in a special way, using the **new** keyword.

This says "Create a new pet object with these property values." The **Pet** constructor function uses the values to create a customized Pet object. So, in our example, we use **Pet** to create two pet objects, **annie** and **willie**, each with different property values.

What's the deal with **this**?

In the **Pet** constructor, we use **this** to mean "this object," and to distinguish properties we're setting from regular variables. If we didn't use **this**, we wouldn't be setting the properties, we'd just be setting variables and those would then not be part of the object created by the constructor (and so, not accessible using the dot notation).

So, when we write **this.type = type**, it's analogous to the **type: "dog"** line in the original code where we wrote out each object literally, only of course when we write **this.type = type** we're setting the value of the *property* **type** to the value of the *parameter* **type**.

Are you wondering why the parameter names are the same as the object's property names? You might think that is confusing. They don't actually need to be the same; that's another convention that helps programmers make sure they're passing in all the right values for setting the properties.

Changing Object Properties

You can change the value of a property in an object after that object's created, simply by assigning it a new value. For instance, suppose you want to change Annie's name to her full name, "Annie Boots" later in your program:

DEVELOPER CONSOLE SESSION:

```
var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
annie.name = "Annie Boots";
```

You can even change the values in the **likes** array property:

DEVELOPER CONSOLE SESSION:

```
annie.likes[2] = "purring";
```

Now, if you use `console.log` to display the value of **annie.likes**, you'll see:

```
> var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
undefined
> annie.name = "Annie Boots";
"Annie Boots"
> annie.likes[2] = "purring";
"purring"
> annie.likes
["sleeping", "teasing pickles", "purring"]
>
```

Built-In Objects

Now that you know about objects, things like **document** and **window** and element objects and **new Array()** are going to make a whole lot more sense, right?

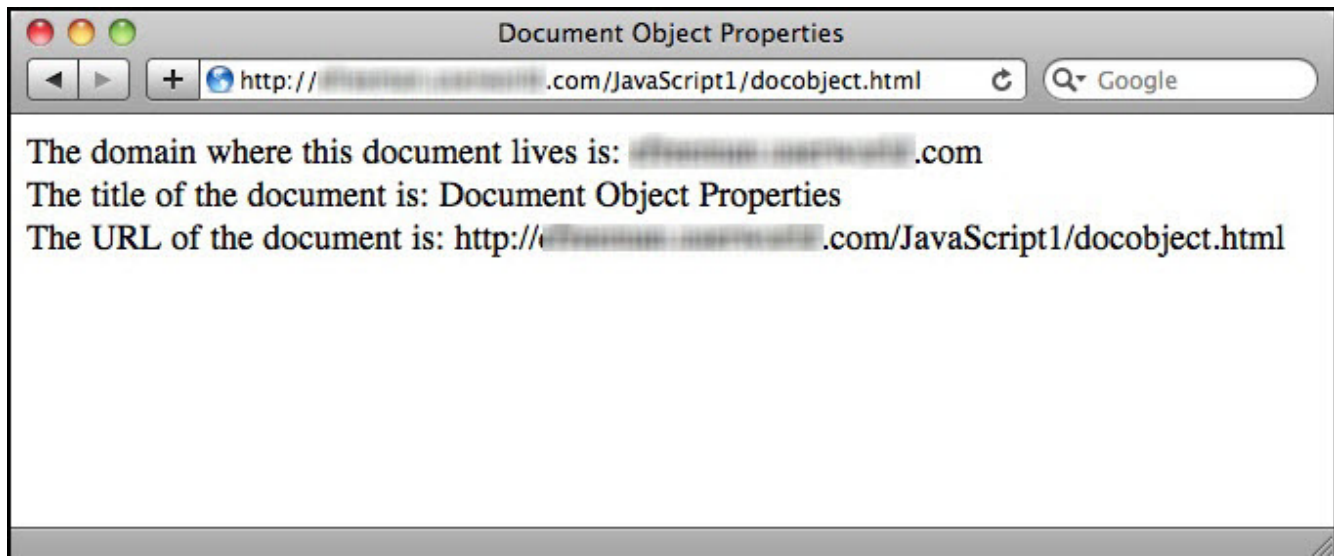
Let's take a look at some of the properties of the *document object*. What do you think the following program will do?

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Document Object Properties </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;
    function init() {
      var div = document.getElementById("result");
      div.innerHTML =
        "The domain where this document lives is: " + document.domain + "<br>" +
        "The title of the document is: " + document.title + "<br>" +
        "The URL of the document is: " + document.URL;
    }
  </script>
</head>
<body>
  <div id="result">
  </div>
</body>
</html>
```

This program uses the *document object*, which is a built-in object in JavaScript (that is, you don't have to create it—it's just there). We're accessing three properties of this object, **document.domain**, **document.title**, and **document.URL**.

Save it in your **/javascript1** folder as **documentobject.html** and click [Preview](#). Did you get the results you expected? Here's what we get (your property values might be slightly different):



What do you think happens if you misspell the id of the object you're trying to retrieve from the document? Let's see:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Document Object Properties </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;
    function init() {
      var div = document.getElementById("resultOops");
      div.innerHTML =
        "The domain where this document lives is: " + document.domain + "<br>" +
        "The title of the document is: " + document.title + "<br>" +
        "The URL of the document is: " + document.URL;
    }
  </script>
</head>
<body>
  <div id="result">
  </div>
</body>
</html>
```

Save and reload the page. Do you see the results you expect? Open up the console and see if you see an error (if you don't, just reload the page). You should see something like this:

OBSERVE:

Uncaught TypeError: Cannot set property 'innerHTML' of null

Because we misspelled the id as "resultOops" instead of "result", we don't actually get back an object from the document, so the variable **div** has nothing in it. The way JavaScript represents "I expected an object in this variable, but I didn't get one" is to put the value **null** in the variable. So **div** contains the value **null**, which indicates we did not get an object back from the call to **document.getElementById("resultOops")**. This is a common mistake (we all make spelling mistakes!), so now you'll know what to watch out for. Go ahead and fix the mistake now, so your code works again.

Element Objects

You've been using element objects already, in quite a few examples. Any time you write something like this, you're using an element object:

OBSERVE:

```
var div = document.getElementById("result");
```

In this case, the variable **div** contains an element object that is returned by calling **document.getElementById("result")**. The element object you get back is the object corresponding to the **<div>** element with the id "result." (Look in the HTML for the example above and you'll see the **<div>**).

Element objects also have properties, and in fact you've been using one of them already:

OBSERVE:

```
div.innerHTML = "The domain where ...";
```

innerHTML is a property that all element objects have. We'll be exploring element objects in more depth later in the course when you learn how to create elements and add them to the DOM!

Using Objects to Collect Global Variables

Before we end this lab, let's do one more thing that we mentioned in the previous lab: look at how you might use an

object to reduce the number of global variables in your programs.

It's pretty easy, actually. Let's say you have three global variables:

- `var random = 0;`
- `var prevValue = 0;`
- `var currentValue = 0;`

and you can't make any of them local variables, because you need them all in multiple functions, and you can't easily pass the variables between functions as arguments.

You can reduce the number of global variables from three to one, simply by collecting these variables inside an object, like this:

OBSERVE:

```
var global = {  
  random: 0,  
  prevValue: 0,  
  currentValue: 0  
}
```

and then, of course, when you need to use one of these global variables, you just write **global.random**, or **global.prevValue**, or **global.currentValue**. One way to make sure your global variables never clash with any other JavaScript you might be including (such as libraries, or other code in separate files) is to name your global object something you know will be unique; for instance, you could use your initials and name it **JTS_global** if your name is "Joe Thomas Smith".

In the next lab, we'll explore how to add behavior to objects. For now, practice creating a few objects of your own.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Methods

Lesson Objectives

When you complete this course, you will be able to:

- create your own methods.
- pass arguments to your methods.

Objects and Methods

You know about objects and properties, and you know that `document.URL` refers to a property named `URL` that is in the object `document`. So what is `document.getElementById()`? We've used this a number of times when we want to **get an element from the DOM** so we can update a web page, like we did in the previous lab:

OBSERVE:

```
window.onload = init;
function init() {
  var div = document.getElementById("result");
  div.innerHTML =
    "The domain where this document lives is: " + document.domain + "<br>" +
    "The title of the document is: " + document.title + "<br>" +
    "The URL of the document is: " + document.URL;
}
```

In this example, we used `document.getElementById("result")` to get the `<div>` element with the id "result" so we could update the page with information from the `document` object's properties.

A Method is a Function in an Object

We say that `getElementById()` is a *method*; that means it's a *function in an object*. In the case of `getElementById()`, this method is in the `document` object. To illustrate how you can create your own methods, why don't we add a method to the Pet constructor? Open `constructor.html` from the last lesson and then use the Save As icon to save it in your `/javascript1` folder as `method.html`. Then, edit it as shown:

CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> Object Methods </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function Pet(type, name, weight, likes) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;

      this.bark = function() {
        return "Woof!";
      };
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting", "eating"]);

      var div = document.getElementById("pets");
      div.innerHTML = annie.name + " is a " + annie.type + " and " + willie.name +
      " is a " + willie.type;
      div.innerHTML = willie.name + " says " + willie.bark();
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click . You should see "Willie says Woof!" in the web page.

We changed the Pet constructor to include a property whose value is a function. The property name is **bark** and the value of the property is a function. You might be thinking that's an odd way to define a function, and you're right, it's different from what we've seen up to now.

Rather than writing **function bark() { ... }**, we write **this.bark = function() { ... }**; which looks a little odd. But this is just another way of defining a function, and it's how you define a method in an object: you're setting the value of the property **bark** to a *function value*. (Notice that, just like for the other properties we assign in the constructor, we end the assignment statement with a ";", so don't forget that.)

Try adding another method, **meow**, to the constructor and test it using **annie.meow()**:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Object Methods </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function Pet(type, name, weight, likes) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;

      this.meow = function() {
        return "Meow!";
      };
      this.bark = function() {
        return "Woof!";
      };
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting", "eating"]);

      var div = document.getElementById("pets");
      div.innerHTML = willie.name + " says " + willie.bark();
      div.innerHTML += "<br>" + annie.name + " says " + annie.meow();
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Notice that when you add methods to objects, you're adding *behavior* to that object. Now the object can *do* something. In this example, now our Pet objects can bark and meow!

Methods with Parameters

So, your Pet objects have a method **bark()**; what if you want to pass arguments to the method? You can add parameters to the method definition, in the parentheses, just like you would in a normal function:

CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> Object Methods </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function Pet(type, name, weight, likes) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;

      this.meow = function() {
        return "Meow!";
      };
      this.bark = function(howMany) {
        var says = "";
        for (var i = 0; i < howMany; i++) {
          says += "Woof! ";
        }
        return says;
      };
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting", "eating"]);

      var div = document.getElementById("pets");
      div.innerHTML = willie.name + " says " + willie.bark(3);
      div.innerHTML += "<br>" + annie.name + " says " + annie.meow();
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click . You should see your page updated with the text "Willie says Woof! Woof! Woof!".

We pass an argument into the pet object's **bark()** method which now has one parameter, **howMany**. We use this parameter to determine how many times the pet should say "Woof!". In this case, we passed in 3, so the **willie** object says "Woof!" three times. Notice that you can declare local variables, and return values from methods, just like you can in a regular function.

This

Suppose you don't like having both **meow()** and **bark()** methods in your Pet objects because you could easily have a situation where you have cats barking and dogs meowing, and that would be bad. You want to replace these methods with another one, **speak()**, and pass in the appropriate sound that a given pet will make when you construct the Pet.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Object Methods </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;


    function Pet(type, name, weight, likes, sound) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;
      this.sound = sound;

      this.meow = function() {
        return "Meow!";
      };
      this.bark = function(howMany) {
        var says = "";
        for (var i = 0; i < howMany; i++) {
          says += "Woof! ";
        }
        return says;
      };

      this.speak = function(howMany) {
        var says = "";
        for (var i = 0; i < howMany; i++) {
          says += this.sound + " ";
        }
        return says;
      };
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"], "Meow");
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting", "eating"], "Woof!");

      var div = document.getElementById("pets");
      div.innerHTML = willie.name + " says " + willie.speak(3);
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click . You should see the same message, "Willie says Woof! Woof! Woof!" in your web page, but now you're using the **speak()** method instead of the **bark()** method. Try changing the code to use the **speak()** method on the annie object.

Notice that we pass in an additional argument to the parameter **sound**, the sound the pet makes ("Meow" for annie, and "Woof!" for willie). As usual, we use `this.sound = sound;` to set the *value of the property **this.sound** to the value of the parameter*.

We deleted the **meow()** and **bark()** methods and replaced them with one method, **speak()**, which does the same thing, except now it uses (with **says += this.sound + " ";**) the **sound** property to create the **says** string that it returns.

Notice that we use **this.sound** to access the property within the function. *This is important!* If you *don't* use **this.sound**, and use just **sound** instead, you're accessing the *parameter sound*, *not* the property of the object! In this case, it wouldn't matter, but what if you don't name your parameters and properties the same? In that case you'll get an error because **sound** would not be a variable. What if you changed the value of the **sound** property after creating the

object? In that case, you won't get the right return value from the **speak()** method. Try this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Object Methods </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function Pet(type, name, weight, likes, sound) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;
      this.sound = sound;

      this.speak = function(howMany) {
        var says = "";
        for (var i = 0; i < howMany; i++) {
          says += this.sound + " ";
        }
        return says;
      };
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"], "Meow");
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting", "eating"], "Woof!");

      var div = document.getElementById("pets");
      div.innerHTML = willie.name + " says " + willie.speak(3);

      annie.sound = "Purrrr";
      div.innerHTML += "<br>" + annie.name + " says " + annie.speak(2);
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it and click **Preview**. You *should* see the message "Purrrr Purrrr" in the page—but because we didn't use **this.sound**, it didn't work correctly! You'll see "Meow Meow" in the page instead.

How does JavaScript know which object "this" is?

When you create a "new" object, like **var willie = new Pet(...)**, JavaScript makes sure that **this** points to that new (willie) object. That way, each new object has its own value for **this** that points to itself. So willie has a **this** that points to willie, and annie has a **this** that points to annie.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

JavaScript and Forms

Lesson Objectives

When you complete this course, you will be able to:

- create a page with a form.
 - add JavaScript to process the form.
 - set up a click handler for the button input control.
 - get the value of a text input control, a numeric input control, a select input control, and a text area control.
 - validate form input and process the data.
 - clear form controls.
-

If you've used or written HTML Forms, you've probably run across JavaScript to *validate* a form: that is, code that checks to see if the values conform to a certain format or range before submitting the form to the back-end server.

You've already seen a couple of examples of using JavaScript with form controls: we've used **`document.getElementById()`** to get the value of a form input control, and we've used a click handler function with the **`onclick`** property to detect when a form button control was clicked. We'll expand on these examples in this lesson, and build a web application you can use to create a list of your favorite movies.

Create a Page with a Form

Start by typing in the HTML for a form that you can use to enter a movie title, a rating, a genre and a description. In addition to the controls for entering the movie data, we'll have a button to click to add a movie to the list. Each of these form controls is a different type, so be careful as you type them in. After you type the HTML, we'll look at the JavaScript you need to get values from each of these types of form controls.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> My Favorite Movies </title>
  <meta charset="utf-8">
  <script>
    // we'll add code here in the next step
  </script>
</head>
<body>
  <h1>Movie List Builder</h1>
  <form>
    <label for="title">Movie title: </label>
    <input id="title" name="title" type="text" size="30" required><br>
    <label for="rating">Rating: </label>
    1 <input id="rating" name="rating" type="range" max="5" min="1"> 5<br>
    <label for="genre">Genre: </label>
    <select id="genre" name="genre">
      <option value="drama" selected>Drama</option>
      <option value="action">Action</option>
      <option value="comedy">Comedy</option>
      <option value="scifi">Sci Fi</option>
      <option value="thriller">Thriller</option>
    </select><br>
    <label for="description">Description:</label><br>
    <textarea id="description" name="description"></textarea>
    <br>
    <input type="button" id="submitButton" value="Add movie!"><br>
  </form>
  <div id="movies">
  </div>
</body>
</html>
```

Save it in your **/javascript 1** folder as **movies.html** and click **Preview** ; it will look like this:

My Favorite Movies

http://.../JavaScript1/movies.html

Movie List Builder

Movie title:

Rating: 1 5

Genre:

Description:

We have four different kinds of form controls here:

OBSERVE:

```
<form>
  <label for="title">Movie title: </label>
  <input id="title" name="title" type="text" size="30" required><br>
  <label for="rating">Rating: </label>
  1 <input id="rating" name="rating" type="range" max="5" min="1"> 5<br>
  <label for="genre">Genre: </label>
  <select id="genre" name="genre">
    <option value="drama" selected>Drama</option>
    <option value="action">Action</option>
    <option value="comedy">Comedy</option>
    <option value="scifi">Sci Fi</option>
    <option value="thriller">Thriller</option>
  </select><br>
  <label for="description">Description:</label><br>
  <textarea id="description" name="description"></textarea>
  <br>
  <input type="button" id="submitButton" value="Add movie!"><br>
</form>
```

- **text input**: you can type in any kind of content.
- **range input**: you're restricted to whole numbers between the min and max specified.
- **select with options**: you must select one of the options. The default is "drama".

- **textarea**: you can type in any kind of content.

In this example, the only input required from the user is the movie title; if the user doesn't change the rating or genre, they'll get the default values of 3 and "drama," and the description can be empty. Using a range type and a select with options makes it easier to validate the form because we know for sure that the user must select a number in the given range (for the range input) and one of the valid options (for the select input). So the text input for the title is really the only unknown factor in this example.

Notice also that we're using a **button** input type, rather than a **submit** input type. That's because we want to get the values from the form using JavaScript rather than submitting the values to the server script straight away. And in this example, we don't actually submit the values to the server script at all; but if you wanted to, you could do that using JavaScript after you've validated and processed all the form data. (We've included the **name** attribute on all the form controls in case you want to try submitting this to a server-side script; if you don't know how to do this, don't worry about it, you don't need to know for this course!)

Add JavaScript to Process the Form

Now it's time to add some JavaScript to process the form data. Notice that we're reusing the **Movie** object from a previous lab that you did. Take a few minutes to review it. We also have an empty array, **movieList**, that we'll use to store all the movies you enter when you test the page.

Go ahead and type this in now and we'll step through the rest of the code after you've tested it.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> My Favorite Movies </title>
  <meta charset="utf-8">
  <script>
    function Movie(title, rating, genre, description) {
      this.title = title;
      this.rating = rating;
      this.genre = genre;
      this.description = description;
    }

    var movieList = [];

    window.onload = init;

    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getMovieData;
    }

    function getMovieData() {
      var titleInput = document.getElementById("title");
      var title = titleInput.value;

      var ratingInput = document.getElementById("rating");
      var rating = parseInt(ratingInput.value);

      var genreSelect = document.getElementById("genre");
      var genreOption = genreSelect.options[genreSelect.selectedIndex];
      var genre = genreOption.value;


      var descriptionTextarea = document.getElementById("description");
      var description = descriptionTextarea.value;

      if (title == null || title == "") {
        alert("Please enter a movie title");
        return;
      }
      else {
        var movie = new Movie(title, rating, genre, description);
        movieList.push(movie);
        var movies = document.getElementById("movies");
        movies.innerHTML = "Added " + movie.title + " to the list.";
      }
    }
  </script>
</head>
<body>
  <h1>Movie List Builder</h1>
  <form>
    <label for="title">Movie title: </label>
    <input id="title" type="text" size="30" required><br>
    <label for="rating">Rating: </label>
    1 <input id="rating" type="range" max="5" min="1"> 5<br>
    <label for="genre">Genre: </label>
    <select id="genre">
      <option value="drama">Drama</option>
      <option value="action">Action</option>
      <option value="comedy">Comedy</option>
      <option value="scifi">Sci Fi</option>
      <option value="thriller">Thriller</option>
    </select><br>
    <label for="description">Description:</label><br>
    <textarea id="description"></textarea>
```

```

    <br>
    <input type="button" id="submitButton" value="Add movie!"><br>
  </form>
  <div id="movies">
  </div>
</body>
</html>

```

Save it and click . Try adding a movie. Add a few movies, trying different values for all the form controls. We agree; it's not that satisfying because we're not actually adding the movies to the page yet, but we're getting there! If you want to see the movies in your **movieList** array, you can always open up the console and type **movieList** at the console prompt. Since **movieList** is a global variable, you can inspect it using the console.



Let's step through the code now.

Set Up a Click Handler for the Button Input Control

Remember from earlier that the first thing we need to do is set up a **click handler** function that will be called when the **Add movie!** button is clicked. We've got that in the **init()** function as usual. When you click the button with the id "submitButton", we call the **getMovieData()** function.

OBSERVE:

```

window.onload = init;

function init() {
  var submitButton = document.getElementById("submitButton");
  submitButton.onclick = getMovieData;
}

```

Get the Value of a Text Input Control

The **getMovieData()** function is where all the real work happens. First, we get the **value** of the **title** text input form control:

OBSERVE:

```
var titleInput = document.getElementById("title");  
var title = titleInput.value;
```

We get the "title" form element from the DOM and store it in the **titleInput** variable. To get the string value from this form element, which is a simple text input, we use the **value** property of the form element. Because this is a required field, we also need to make sure the user entered something, which we do later in the function:

OBSERVE:

```
if (title == null || title == "") {  
    alert("Please enter a movie title");  
    return;  
}
```

There, we **check to see if title is null OR the empty string ""**, and if it is, we **alert the user to enter a movie title** and **return**. The return ends the function, so nothing else happens until the user enters a title.

A Slight Sidetrack: Logical Operators

If you're paying close attention you probably noticed the **||** in the code:

OBSERVE:

```
if (title == null || title == "") {  
    alert("Please enter a movie title");  
    return;  
}
```

and you're wondering what that is. This is a *logical operator* and it means "or." This operator allows you to test two things at once: if the first test is true, then the whole conditional expression is true. Same for the second test; if the second test is true, then the whole conditional expression is true. Likewise if *both* tests are true, then the whole conditional expression is true. The only case where the conditional expression is false is if both tests are false.

Another logical operator you'll use frequently is **&&**, which means "and." In this case, *both* conditional tests must be true! So, you might use it like this:

```
if (color != null && color == "blue") { ... }
```

In this case it must be true that **color** is not null, **AND** that **color** equals "blue" in order for the conditional expression to be true.

There are other logical operators, but these are the two you'll use most frequently.

Get the Value of a Numerical Input Control

Back up a bit in the function, below where we just got the **movie title**, we **get the rating**. The rating input is a *range* input control, so we don't have to worry about this input value being empty (like we did for the movie title), but we do need to convert the value to a number (not all browsers do this automatically for this input type yet). We can use the **parseInt()** function to convert the value we get from the range into an integer number. In this example, we don't actually need the value as a number, but if you want to, say, sort the movies by their rating, it would be handy to have this value as a number rather than a string.

OBSERVE:

```
var ratingInput = document.getElementById("rating");  
var rating = parseInt(ratingInput.value);
```

Notice that we're again using the **value** property of the form element to get the value of the range input, and then using the **parseInt()** function to convert that value to an integer to store in the **rating** variable.

Note

If you want your code to work in IE9 or earlier, you will need to use a text input control instead. The range type was added in HTML5 and IE didn't support it until version 10. If you use a text input, you'll need to make sure the user entered a number. Remember that you can use `isNaN()` to check for a number.

Get the Value of a Select Input Control

Next, we get the value of the select input control. This takes an extra step because we need to figure out which option in the select was selected. We do this using the `selectedIndex` property of the `<select>` element object, and then use that index to get the `selected option` from the options array.

OBSERVE:

```
var genreSelect = document.getElementById("genre");
var genreOption = genreSelect.options[genreSelect.selectedIndex];
var genre = genreOption.value;
```

Let's walk through an example so you can see how this code to get the selected option works. Suppose you choose "Comedy" from the genre menu. The `genreSelect` variable holds the entire `<select>` element, and a `<select>` element has a special property, `options`, that contains an array of all the options in the `<select>`. In our case, that array will contain five values, the various movie genres. To find out which item is selected, use the `genreSelect.selectedIndex` property. You selected "Comedy," the third item in the list, which means it's got index 2 in the array of `options` (because arrays start with an index of 0), so `genreSelect.selectedIndex` equals 2. `genreSelect.options[2]` is the option you selected, and we store that in the `genreOption` variable. We can then use the `value` property to get the value of the option, which results in the string "Comedy" being stored in the `genre` variable.

Get the Value of a Text Area Control

Finally, we get the `value of the textarea control`. This is very similar to getting the value of a text input control.

OBSERVE:

```
var descriptionTextarea = document.getElementById("description");
var description = descriptionTextarea.value;
```

Validate Form Input and Process the Data

Now that we've got all the data from the form, we can check the data. As we noted above, the only input we need to check is the "title" text input, and all we're going to do in this example is make sure it's not `null or empty`.

OBSERVE:

```
if (title == null || title == "") {
    alert("Please enter a movie title");
    return;
}
else {
    var movie = new Movie(title, rating, genre, description);
    movieList.push(movie);
    var movies = document.getElementById("movies");
    movies.innerHTML = "Added " + movie.title + " to the list.";
}
```

If the `"title" input data is okay`, we create a new `movie` object using the `Movie` constructor, and pass in the four pieces of data we retrieved from the form: title, rating, genre, and description.

If the user didn't enter a description, then the `description` argument will be the empty string, which is fine. Once we've created the new `movie` object, we add it to the `movieList` array using the array `push()` method, which just appends the item to the end of the array.

Finally, we update the content of the `movies <div>` to display the title of the movie we just added.

Clearing Form Controls

After you click the button to add a movie to the list, all the values in the form controls stay in the form controls, so to add another movie, you need to change the values? That can be a pain... so let's *reset* the form. You might have done this previously by adding a Reset button to your HTML, like this:

```
<input type="reset" value="Reset">
```

You can do a reset using JavaScript instead. The best time to do it is after you've collected all the form data and made sure it's correct. So update the **getMovieData()** function to add this code at the bottom, in the **else** clause. And don't forget to add an id "theForm" to the form element in the HTML.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> My Favorite Movies </title>
  <meta charset="utf-8">
  <script>
    function Movie(title, rating, genre, description) {
      this.title = title;
      this.rating = rating;
      this.genre = genre;
      this.description = description;
    }

    var movieList = [];

    window.onload = init;

    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getMovieData;
    }

    function getMovieData() {
      var titleInput = document.getElementById("title");
      var title = titleInput.value;

      var ratingInput = document.getElementById("rating");
      var rating = parseInt(ratingInput.value);

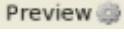
      var genreSelect = document.getElementById("genre");
      var genreOption = genreSelect.options[genreSelect.selectedIndex];
      var genre = genreOption.value;

      var descriptionTextarea = document.getElementById("description");
      var description = descriptionTextarea.value;

      if (title == null || title == "") {
        alert("Please enter a movie title");
        return;
      }
      else {
        var movie = new Movie(title, rating, genre, description);
        movieList.push(movie);
        var movies = document.getElementById("movies");
        movies.innerHTML = "Added " + movie.title + " to the list.";

        var theForm = document.getElementById("theForm");
        theForm.reset();
      }
    }
  </script>
</head>
<body>
  <h1>Movie List Builder</h1>
  <form id="theForm">
    <label for="title">Movie title: </label>
    <input id="title" type="text" size="30" required><br>
    <label for="rating">Rating: </label>
    1 <input id="rating" type="range" max="5" min="1"> 5<br>
    <label for="genre">Genre: </label>
    <select id="genre">
      <option value="drama">Drama</option>
      <option value="action">Action</option>
      <option value="comedy">Comedy</option>
      <option value="scifi">Sci Fi</option>
      <option value="thriller">Thriller</option>
    </select>
  </form>
</body>
</html>
```

```
</select><br>
<label for="description">Description:</label><br>
  <textarea id="description"></textarea>
<br>
<input type="button" id="submitButton" value="Add movie!"><br>
</form>
<div id="movies">
</div>
</body>
</html>
```

Save it and click . Try entering a movie. The form resets after you click **Add movie!** Try adding a few more movies.

Notice that we only reset the form when we've successfully got all the data. If the user enters other values, but forgets the title, we don't want to reset in that case, because that would force them to re-enter those other values. It's a good idea to consider carefully where in your code you reset a form.

And we're done! In the next lesson, you'll learn about form collections and how to submit a form.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Form Collections

Lesson Objectives

When you complete this course, you will be able to:

- use form collections.
 - use the elements collection to access the value of a selected radio button.
 - submit a form with JavaScript.
-

In this lesson, we temporarily leave our movie application to explore the **forms** collection. But don't worry, we'll come back to movies in the next lesson!

Using Form Collections

The document object contains an object named **forms** that contains all the forms in your page. This is known as the *forms collection*. So, instead of accessing your form like this:

```
var theForm = document.getElementById("theForm");
```

you can access your form like this:

```
var theForm = document.forms.theForm;
```

Let's try using the forms collection now. Take a moment to review the HTML as you're typing it in and notice the different kinds of form inputs we're using: text, checkbox, textarea, date, and of course a button to submit the values.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Personality Quiz </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getData;
    }

    function getData() {
      var theForm = document.forms.theForm;
      var color = theForm.elements.color.value;

      var results = document.getElementById("results");
      results.innerHTML = "Favorite color: " + color;
    }
  </script>
</head>
<body>
  <h1>Personality Quiz</h1>
  <form id="theForm">
    <p>
      <label for="color">What is your favorite color?</label>
      <input type="text" id="color" placeholder="blue" required>
    </p>
    <p>Are you...</p>
    <input type="checkbox" id="funny" name="personality" value="funny">
      <label for="funny">funny</label>
    <input type="checkbox" id="smart" name="personality" value="smart">
      <label for="smart">smart</label>
    <input type="checkbox" id="shy" name="personality" value="shy">
      <label for="shy">shy</label>
    <input type="checkbox" id="outgoing" name="personality" value="outgoing">
      <label for="outgoing">outgoing</label>
    <br>
    <p>Tell us a little about yourself:</p>
    <textarea id="aboutme" name="textarea" rows="12" cols="38">
    </textarea>
    <br>
    <label for="date">What's your birthday?</label>
    <input id="date" name="date" type="date">
    <input id="submitButton" type="button" value="Submit">
  </form>
  <h2>Results</h2>
  <div id="results">
  </div>
</body>
</html>
```

Notice that we're using an attribute you haven't seen before on some of these elements; the **name** attribute. This attribute is required for form elements that need to be grouped together, like checkboxes that are all associated with personality type, as in the example above. We won't need to use the **name** attribute in our code, but it's a good habit to use it, especially when you start processing forms on the server side (e.g. with a PHP script).

Save this code in your **/javascript1** folder as **personality.html** and click . Enter a color and click **Submit**. Your favorite color appears below **Results** at the bottom of the page:

Personality Quiz

What is your favorite color?

Are you...

☐ funny ☐ smart ☐ shy ☐ outgoing

Tell us a little about yourself:

What's your birthday?

Results

Favorite color: pink

Just like before, when we used `document.getElementById` to get the form, we use the id of the form **"theForm"** to access the form. However, using the *forms collection*, we use the **forms** object, which is a property of the **document** object. As you know, to access a property of an object, you use dot notation. So **document.forms** is an object containing all the forms in the page, and that object contains a property, **theForm**, which is the form with the id "theForm," which is the form we want.

OBSERVE:

```
var theForm = document.forms.theForm;
```

The **form itself is an element object**. One of its properties is another collection object, the *elements collection*. This is an object that contains all the elements contained in the form. It is similar to the forms collection we just looked at. So you can use it to access elements in the form, like this:

OBSERVE:

```
var color = theForm.elements.color.value;
```

To access the **elements collection**, we again use dot notation. And just like before, because each form control is a property of the **elements** object, we can use dot notation to access a **control using its id**. So, to access the control with the id "color," we write **theForm.elements.color**.

We combined two steps here: getting access to the control and **getting access to the value**. This is perfectly okay as long as you're sure that the form control exists. If it doesn't exist, you'll get an error because the value of **theForm.elements.color** will be null! In this case, however, we're sure that the **color** control exists, so we won't have that problem.

Note

If you need a refresher on element objects and the document object model, review the [Document Object Model lesson](#).

Let's get the "About Me" and "Birthday" parts of the form using the forms and elements collections too:


CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Personality Quiz </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getData;
    }

    function getData() {
      var theForm = document.forms.theForm;
      var color = theForm.elements.color.value;
      var aboutme = theForm.elements.aboutme.value;
      var birthday = theForm.elements.date.value;

      var results = document.getElementById("results");
      results.innerHTML = "Favorite color: " + color + "<br>";
      results.innerHTML += "About me: " + aboutme + "<br>";
      results.innerHTML += "Birthday: " + birthday + "<br>";
    }
  </script>
</head>
<body>
  <h1>Personality Quiz</h1>
  <form id="theForm">
    <p>
      <label for="color">What is your favorite color?</label>
      <input type="text" id="color" placeholder="blue" required>
    </p>
    <p>Are you...</p>
    <input type="checkbox" id="funny" name="personality" value="funny">
      <label for="funny">funny</label>
    <input type="checkbox" id="smart" name="personality" value="smart">
      <label for="smart">smart</label>
    <input type="checkbox" id="shy" name="personality" value="shy">
      <label for="shy">shy</label>
    <input type="checkbox" id="outgoing" name="personality" value="outgoing">
      <label for="outgoing">outgoing</label>
    <br>
    <p>Tell us a little about yourself:</p>
    <textarea id="aboutme" name="textarea" rows="12" cols="38">
    </textarea>
    <br>
    <label for="date">What's your birthday?</label>
    <input id="date" name="date" type="date">
    <input id="submitButton" type="button" value="Submit">
  </form>
  <h2>Results</h2>
  <div id="results">
  </div>
</body>
</html>
```

Save it and click . You should get the same results as before.

Using the Elements Collection to Access the Value of a Selected Radio Button

The elements collection is very handy when you want to get the value of a radio button or checkbox from a form. Let's use the forms collection to get the personality items from the form.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Personality Quiz </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

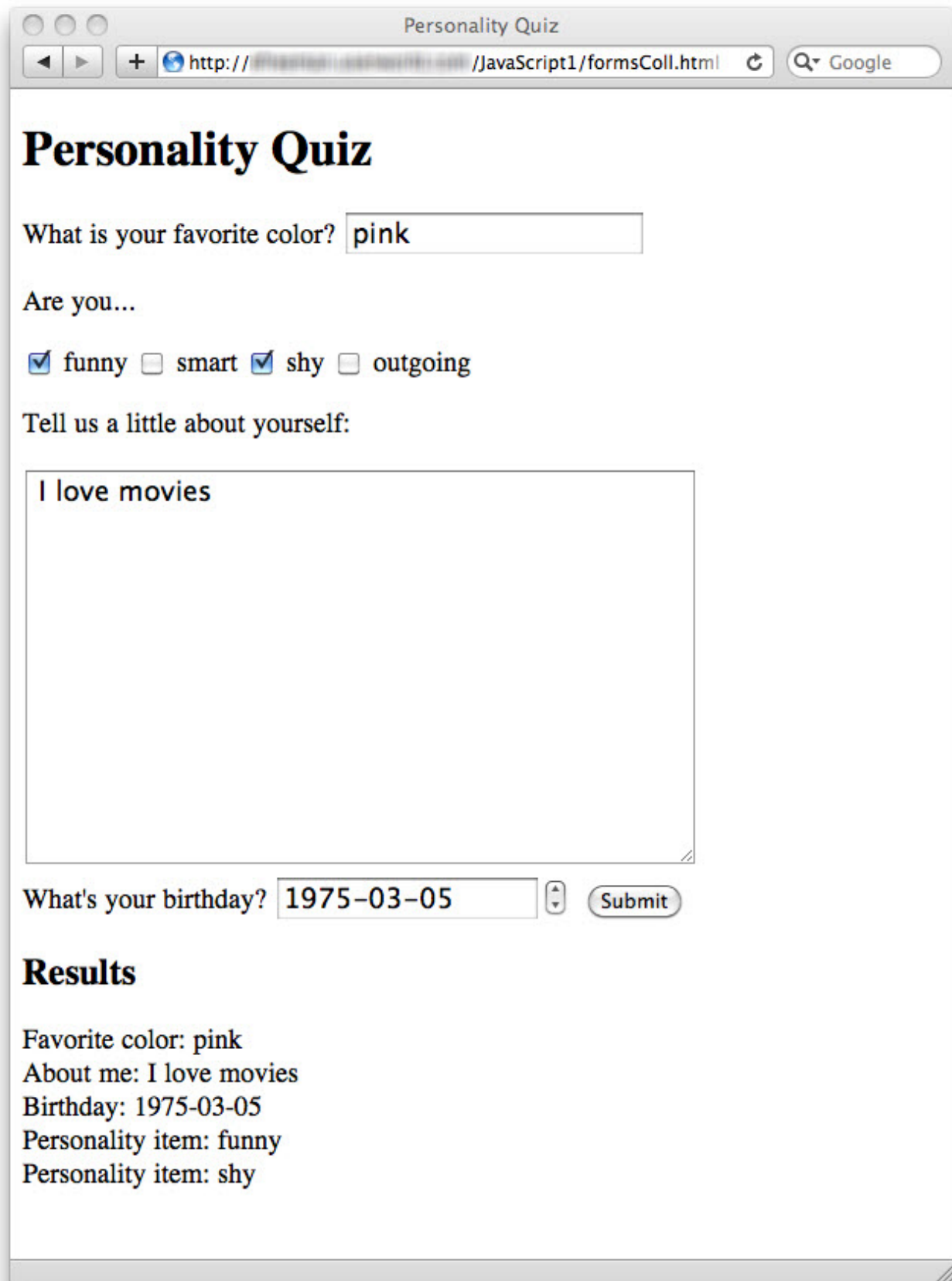
    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getData;
    }

    function getData() {
      var theForm = document.forms.theForm;
      var color = theForm.elements.color.value;
      var aboutme = theForm.elements.aboutme.value;
      var birthday = theForm.elements.date.value;

      var results = document.getElementById("results");
      results.innerHTML = "Favorite color: " + color + "<br>";
      results.innerHTML += "About me: " + aboutme + "<br>";
      results.innerHTML += "Birthday: " + birthday + "<br>";

      var personalityArray = theForm.elements.personality;
      for (var i = 0; i < personalityArray.length; i++) {
        if (personalityArray[i].checked) {
          results.innerHTML += "Personality item: " + personalityArray[i].value + "<br>";
        }
      }
    }
  </script>
</head>
<body>
  <h1>Personality Quiz</h1>
  <form id="theForm">
    <p>
      <label for="color">What is your favorite color?</label>
      <input type="text" id="color" placeholder="blue" required>
    </p>
    <p>Are you...</p>
    <input type="checkbox" id="funny" name="personality" value="funny">
      <label for="funny">funny</label>
    <input type="checkbox" id="smart" name="personality" value="smart">
      <label for="smart">smart</label>
    <input type="checkbox" id="shy" name="personality" value="shy">
      <label for="shy">shy</label>
    <input type="checkbox" id="outgoing" name="personality" value="outgoing">
      <label for="outgoing">outgoing</label>
    <br>
    <p>Tell us a little about yourself:</p>
    <textarea id="aboutme" name="textarea" rows="12" cols="38">
    </textarea>
    <br>
    <label for="date">What's your birthday?</label>
    <input id="date" name="date" type="date">
    <input id="submitButton" type="button" value="Submit">
  </form>
  <h2>Results</h2>
  <div id="results">
  </div>
</body>
</html>
```


Save it and click [Preview](#). Try checking one or more of the "Are you..." boxes. When you submit the form, the items you checked appear under **Results** at the bottom of the page:



The screenshot shows a web browser window titled "Personality Quiz". The address bar shows the URL "http://.../JavaScript1/formsColl.html". The page content includes a title "Personality Quiz", a text input field for "What is your favorite color?" with the value "pink", a section "Are you..." with four checkboxes: "funny" (checked), "smart" (unchecked), "shy" (checked), and "outgoing" (unchecked). Below this is a text area for "Tell us a little about yourself:" containing the text "I love movies". At the bottom of the form is a date input field for "What's your birthday?" with the value "1975-03-05" and a "Submit" button. Below the form, the "Results" section displays the following information: "Favorite color: pink", "About me: I love movies", "Birthday: 1975-03-05", "Personality item: funny", and "Personality item: shy".

Personality Quiz

What is your favorite color?

Are you...

☒ funny ☐ smart ☒ shy ☐ outgoing

Tell us a little about yourself:

What's your birthday?

Results

Favorite color: pink
About me: I love movies
Birthday: 1975-03-05
Personality item: funny
Personality item: shy

Let's step through how this works. First, notice that each checkbox item has a different id, but has the *same name*, **personality**. We don't actually use the ids in this example, but we use the name to get the checkboxes as a group. HTML knows that if the checkboxes have the same name, then they are grouped together. It's also a good reminder

that ids must be unique, but that the name must be the same for all checkbox (or radio button, if you're using that) items in the same group. Also notice that each checkbox item has a different **value**; the string in the value property is what we get when we get the value of the "checked" items.

In the JavaScript code, we use the *elements* collection to **get the personality checkbox group from the form**:

OBSERVE:

```
var personalityArray = theForm.elements.personality;
```

In this case, the **personalityArray** is an collection of checkbox elements, all with the **name** "personality." To see which checkboxes are selected, we loop through the collection and test to see if the item is **checked**:

OBSERVE:

```
for (var i = 0; i < personalityArray.length; i++) {  
    if (personalityArray[i].checked) {  
        results.innerHTML += "Personality item: " + personalityArray[i].value + "<br>";  
    }  
}
```

When we find one that's **checked**, we update the **"results" <div>** to display that checkbox's value. We get the value using the **value** property, as usual; but notice that we are getting the value of the **checkbox item in the collection**.

You can use a similar technique to find which radio button values are selected in a form; however, remember that radio buttons can only have one value selected at a time, while checkboxes can have multiple values selected.

Submitting a Form with JavaScript

Earlier, we said that we used a button control in the form, rather than a submit control, so we could process the form before submitting the form to a server-side script. And in fact, in this example, we don't submit the form at all. However, there may be times when you want to validate form data, and then submit a form to a server-side script.

If you want to be able to submit a form once you've used JavaScript to validate or process the data in the form, you'd only need to make a few changes to your code:

OBSERVE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Personality Quiz </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getData;
    }

    function getData() {
      var theForm = document.forms.theForm;
      var color = theForm.elements.color.value;
      var aboutme = theForm.elements.aboutme.value;
      var birthday = theForm.elements.date.value;

      var results = document.getElementById("results");
      results.innerHTML = "Favorite color: " + color + "<br>";
      results.innerHTML += "About me: " + aboutme + "<br>";
      results.innerHTML += "Birthday: " + birthday + "<br>";

      var personalityArray = theForm.elements.personality;
      for (var i = 0; i < personalityArray.length; i++) {
        if (personalityArray[i].checked) {
          results.innerHTML += "Personality item: " + personalityArray[i].value +
" <br>";
        }
      }

      theForm.submit();
      return false;
    }
  </script>
</head>
<body>
  <h1>Personality Quiz</h1>
  <form id="theForm" method="get" action="http://yourserver.com/yourscript.php">
    <p>
      <label for="color">What is your favorite color?</label>
      <input type="text" id="color" placeholder="blue" required>
    </p>
    <p>Are you...</p>
    <input type="checkbox" id="funny" name="personality" value="funny">
      <label for="funny">funny</label>
    <input type="checkbox" id="smart" name="personality" value="smart">
      <label for="smart">smart</label>
    <input type="checkbox" id="shy" name="personality" value="shy">
      <label for="shy">shy</label>
    <input type="checkbox" id="outgoing" name="personality" value="outgoing">
      <label for="outgoing">outgoing</label>
    <br>
    <p>Tell us a little about yourself:</p>
    <textarea id="aboutme" name="textarea" rows="12" cols="38">
    </textarea>
    <br>
    <label for="date">What's your birthday?</label>
    <input id="date" name="date" type="date">
    <input id="submitButton" type="button" value="Submit">
  </form>
  <h2>Results</h2>
  <div id="results">
  </div>
</body>
```

```
</html>
```

Submitting the form in this example doesn't make much sense, and we don't have a server side script to submit it to anyway, so just take a look at how this would work in case you need to know how to do it.

Notice that all we do is get the form, and then call the form's **submit()** method. We return false from the function so that the JavaScript knows there's nothing else to do after the form has been submitted. That's it!

In the next lesson, you'll learn how to create new elements to add to the DOM for your page and add data you get from a form to a web page, dynamically! See you there!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Creating New Elements

Lesson Objectives

When you complete this course, you will be able to:

- create new elements and add them to the DOM.
 - insert new elements where you want them.
-

When we left our Movie application earlier, you had a web application that you could use to create a list of movies, but no good way to add those movies to the web page. We used **innerHTML** to update the content of a `<div>` element, but what we'd *really* like is a way to add new elements to the web page. In this example, it would be much better if we could create a real list of movies, using the `` and `` elements.

Creating and Adding New Elements to the DOM

You can indeed create new elements using the **document** method **createElement()**, and add these new elements to a web page using the element object methods **appendChild()** and **insertBefore()**. First, let's look at how to use **appendChild()**. Take a look at the code below; we've updated the Movie code from the previous lesson, so make your changes carefully (or start a whole new file) and then we'll take a closer look at the code.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> My Favorite Movies, Take Two </title>
  <meta charset="utf-8">
  <script>
    function Movie(title, rating, genre, description) {
      this.title = title;
      this.rating = rating;
      this.genre = genre;
      this.description = description;

      this.print = function() {
        var s = this.title + "; rated: " + this.rating + "; genre: " + this.genre +
          "; " + this.description;
        return s;
      }
    }

    var movieList = [];

    window.onload = init;

    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getMovieData;
    }

    function getMovieData() {
      var titleInput = document.getElementById("title");
      var title = titleInput.value;

      var ratingInput = document.getElementById("rating");
      var rating = parseInt(ratingInput.value);

      var genreSelect = document.getElementById("genre");
      var genreOption = genreSelect.options[genreSelect.selectedIndex];
      var genre = genreOption.value;

      var descriptionTextarea = document.getElementById("description");
      var description = descriptionTextarea.value;

      if (title == null || title == "") {
        alert("Please enter a movie title");
        return;
      }
      else {
        var movie = new Movie(title, rating, genre, description);
        movieList.push(movie);
var movies = document.getElementById("movies");
movies.innerHTML += "Added " + movie.title + " to the list."
        addMovieToList(movie);

        var theForm = document.getElementById("theForm");
        theForm.reset();
      }
    }

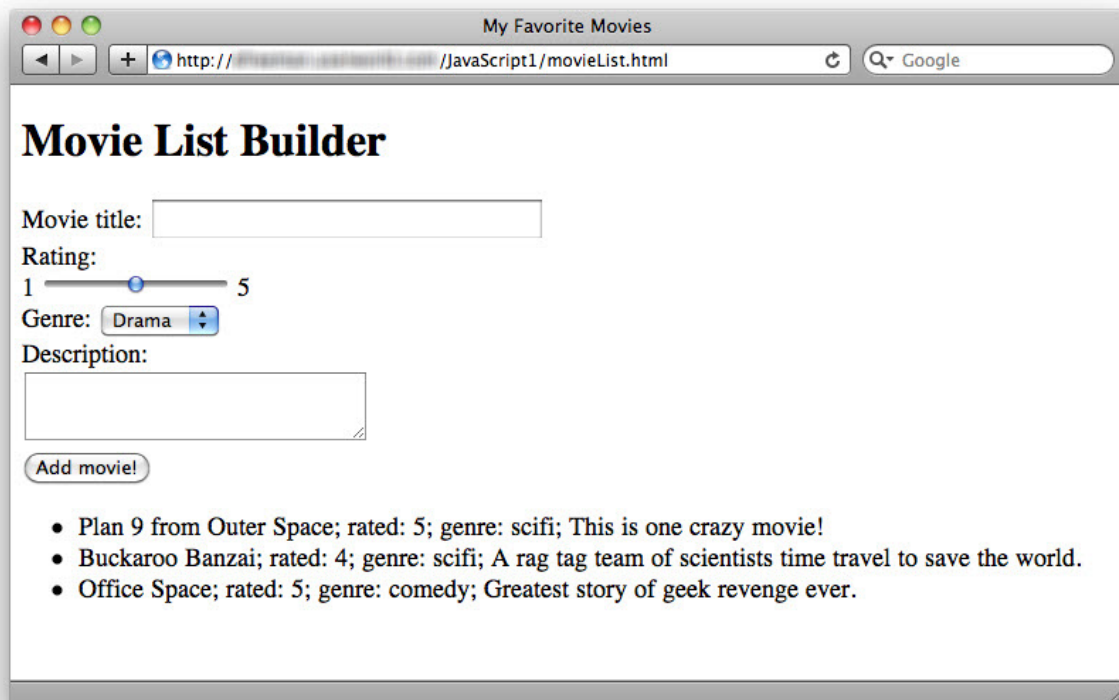
    function addMovieToList(movie) {
      var movieList = document.getElementById("movieList");
      var li = document.createElement("li");
      li.innerHTML = movie.print();
      movieList.appendChild(li);
    }
  </script>
```

```

</head>
<body>
  <h1>Movie List Builder</h1>
  <form id="theForm">
    <label for="title">Movie title: </label>
    <input id="title" name="title" type="text" size="30" required><br>
    <label for="rating">Rating: </label>
    1 <input id="rating" name="rating" type="range" max="5" min="1"> 5<br>
    <label for="genre">Genre: </label>
    <select id="genre" name="genre">
      <option value="drama" selected>Drama</option>
      <option value="action">Action</option>
      <option value="comedy">Comedy</option>
      <option value="scifi">Sci Fi</option>
      <option value="thriller">Thriller</option>
    </select><br>
    <label for="description">Description:</label><br>
    <textarea id="description" name="description"></textarea>
    <br>
    <input type="button" id="submitButton" value="Add movie!"><br>
  </form>
  <div id="movies">
</div>
  <ul id="movieList">
  </ul>
</body>
</html>

```

Save it and click . Try adding a few movies. They should appear below the form in a list:



It's much more satisfying to see your movies show up in the web page, don't you agree?! So how does this work? Let's take a closer look.

First, notice that we've removed the "movies" <div>, and replaced it with an empty "movieList" unordered list, . This is where we'll add the movies to the page.

Next, we added a **print()** method to the Movie object. This method returns a string that contains information about the movie. We use that string in the **addMovieToList()** function; we'll get to that in a moment.

Next, in **getMovieData()**, instead of just updating a <div> with a message, we call a new function,

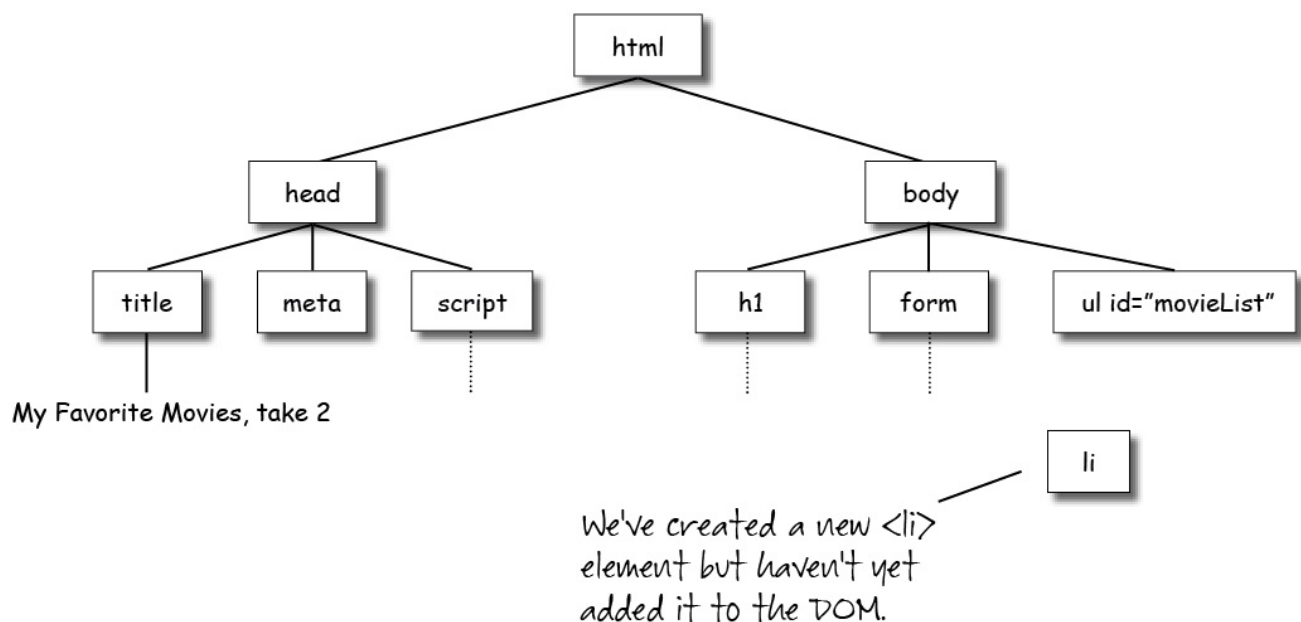
addMovieToList(), passing the new movie object we created.

OBSERVE:

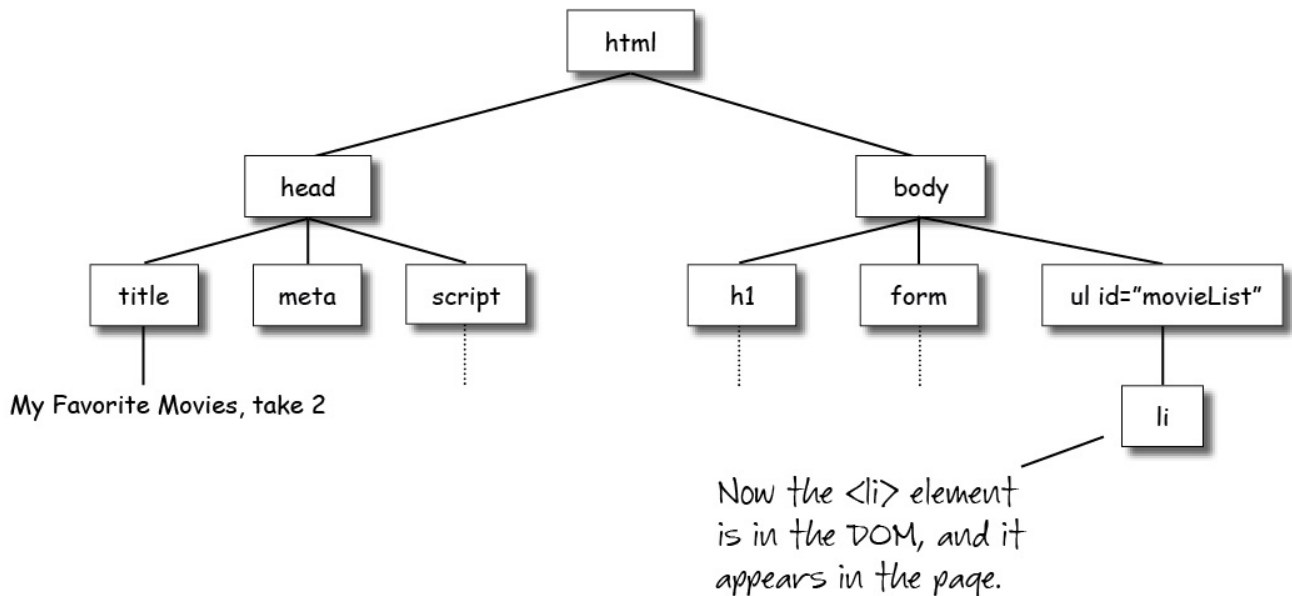
```
function addMovieToList(movie) {  
  var movieList = document.getElementById("movieList");  
  var li = document.createElement("li");  
  li.innerHTML = movie.print();  
  movieList.appendChild(li);  
}
```

In the **addMovieToList()** function, we **create a new element** to add to the list, using the **document.createElement()** method. We pass in **a string with the tag name of the element we want to create**, and we use the trusty **innerHTML** property to set the content of that **** element; in this case, we set the content to the string that is returned from **the movie object's print()** method.

So, we have an **** element, with the content we want, but it's just hanging out there. It's not actually part of the page until we add it to the DOM, so you won't see it until you add it.



To do that, we use the **appendChild()** method and **add it as a child of the "movieList" element**. As soon as we call **appendChild()**, the **** element, with the movie information, appears like magic in our web page!

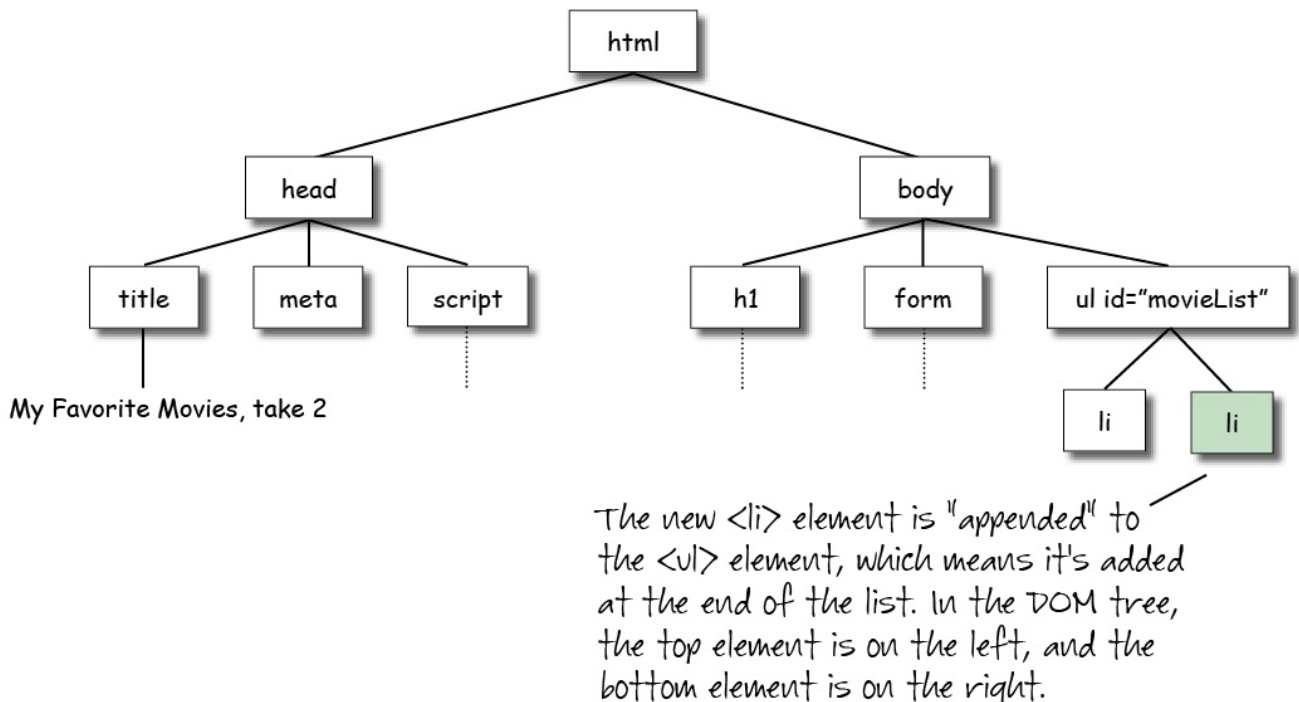


Notice (in `movieList.appendChild(li)`) that the `appendChild()` method is a method of the "movieList" `` element object.

Every element has this method (so you can add a "child" element to any element using this method). You pass in the element you're adding as a child as the argument to the method; in this case, we pass in the new `` element we just created. The end result is a DOM that contains a new `` element (just as if you'd typed it in the HTML!). Remember, the enclosing element (in this case, the `` element) is the *parent* element, and the elements enclosed within the parent are the *child* elements (in this case, the `` elements that we are adding with the JavaScript code). In the DOM tree, it's just like a family tree: all the elements directly below an element object in the tree are that element's children.

Adding More Elements

Try adding another movie. Notice that each new movie you add is added at the *end* of the list. Let's take a look behind the scenes when you add a movie to the page and another `` gets added to the DOM:



The second (and third, and so on) movie you add is *appended* to the list because you are using

`movieList.appendChild(li).`

`appendChild()` adds the new element to the end of the list. So, what happens if you want to add a new element to the *beginning* of the list instead?

Inserting Elements

To add a child element above the other child elements in the parent element (to the *left* of the other child elements in the DOM tree), use the `insertBefore()` method.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> My Favorite Movies, take 2 </title>
  <meta charset="utf-8">
  <script>
    function Movie(title, rating, genre, description) {
      this.title = title;
      this.rating = rating;
      this.genre = genre;
      this.description = description;

      this.print = function() {
        var s = this.title + "; rated: " + this.rating + "; genre: " + this.genre +
          "; " + this.description;
        return s;
      }
    }

    var movieList = [];

    window.onload = init;

    function init() {
      var submitButton = document.getElementById("submitButton");
      submitButton.onclick = getMovieData;
    }

    function getMovieData() {
      var titleInput = document.getElementById("title");
      var title = titleInput.value;

      var ratingInput = document.getElementById("rating");
      var rating = parseInt(ratingInput.value);

      var genreSelect = document.getElementById("genre");
      var genreOption = genreSelect.options[genreSelect.selectedIndex];
      var genre = genreOption.value;

      var descriptionTextarea = document.getElementById("description");
      var description = descriptionTextarea.value;

      if (title == null || title == "") {
        alert("Please enter a movie title");
        return;
      }
      else {
        var movie = new Movie(title, rating, genre, description);
        movieList.push(movie);
        addMovieToList(movie);
        var theForm = document.getElementById("theForm");
        theForm.reset();
      }
    }

    function addMovieToList(movie) {
      var movieList = document.getElementById("movieList");
      var li = document.createElement("li");
      li.innerHTML = movie.print();
      movieList.appendChild(li);
      if (movieList.childElementCount == 0) {
        movieList.appendChild(li);
      }
      else {
        movieList.insertBefore(li, movieList.firstChild);
      }
    }
  </script>
</head>
</html>
```

```

    }
  </script>
</head>
<body>
  <h1>Movie List Builder</h1>
  <form id="theForm">
    <label for="title">Movie title: </label>
    <input id="title" name="title" type="text" size="30" required><br>
    <label for="rating">Rating: </label>
    1 <input id="rating" name="rating" type="range" max="5" min="1"> 5<br>
    <label for="genre">Genre: </label>
    <select id="genre" name="genre">
      <option value="drama" selected>Drama</option>
      <option value="action">Action</option>
      <option value="comedy">Comedy</option>
      <option value="scifi">Sci Fi</option>
      <option value="thriller">Thriller</option>
    </select><br>
    <label for="description">Description:</label><br>
    <textarea id="description" name="description"></textarea>
    <br>
    <input type="button" id="submitButton" value="Add movie!"><br>
  </form>
  <ul id="movieList">
  </ul>
</body>
</html>

```

Save it and click . Try adding a few movies. Your new movies are added to the top of the list.

Look at the code where we add the new element using **insertBefore()**:

OBSERVE:

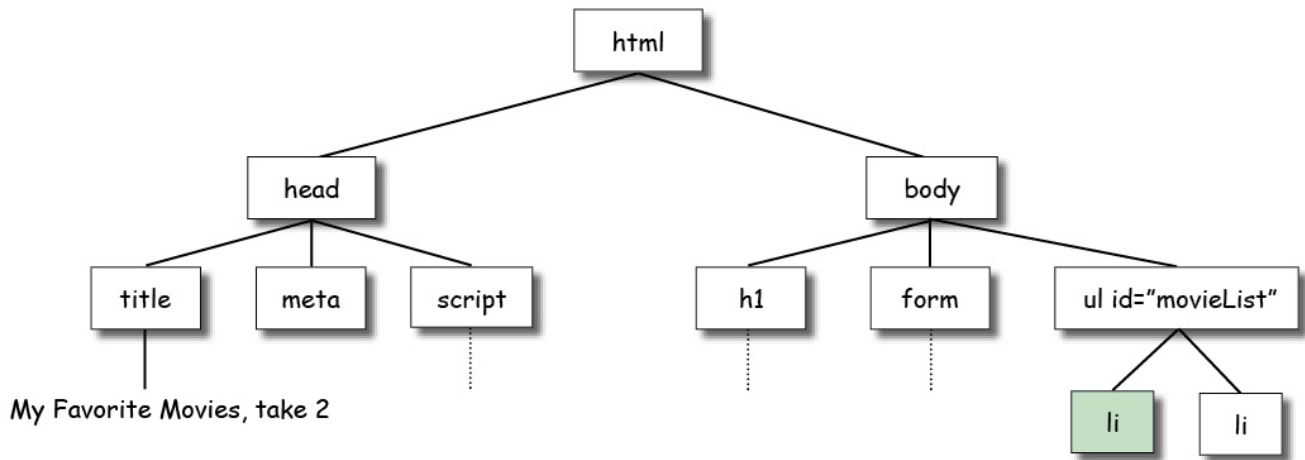
```

if (movieList.childElementCount == 0) {
  movieList.appendChild(li);
}
else {
  movieList.insertBefore(li, movieList.firstChild);
}

```

insertBefore() is (again) a method of the parent element, in this case, the "movieList" element, and that it takes two arguments: the first is the new element you're adding, and the second is the existing **first child** in the list. That means we can use **insertBefore** only if there is already a child element in the "movieList" list.

So, we first **check to see if the list has no children** using the element object property **childElementCount**. If the movieList has zero children, then we need to add the first child using **appendChild()** as we did before. But, if the movieList already has children, then we can use **insertBefore()** to add the new element *before* the first element (which means the new element now becomes the first child). We use the property **firstChild** to get the first child of the parent element. Here's how it looks in the DOM tree:



Using `insertBefore()`, the new `` element is "inserted before" the first child in the `` element; that is, before the first existing `` element. So it is added at the top of the list.

Now you know how to add elements to the DOM. Try adding other kinds of elements. For instance, try making a page with a `<div>` element and add some `<p>` elements to it.

How would you add an `` element? Can you think of a missing piece you need to properly add this element to the page that we haven't talked about yet? You'll find out what the missing piece is and how to add it using JavaScript in the next lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Element Attributes and Style

Lesson Objectives

When you complete this course, you will be able to:

- build the photo viewer application.
 - use `setAttribute` to add a class to an element.
 - use `getAttribute` to get the value of an element attribute.
 - use `document.querySelectorAll()` and `setAttribute()` to set the class of multiple elements.
 - set style directly using JavaScript.
-

One way you can use JavaScript to make your web applications more fun and interesting is to manipulate how your elements look on the page. In the next two lessons, we'll build a photo viewer web application. In this lesson, you'll learn how to use JavaScript to add and remove CSS style on your elements so you can show and hide menus. In the next lesson, you'll learn how to manipulate the DOM more so you can add and remove `` elements on your page. Sound like fun? Let's do it!

First, check out the app you're going to build: [The Photo Viewer App](#).

Building the Photo Viewer Application

Start by creating the HTML for the Photo Viewer App. We'll put the JavaScript and CSS in separate files for this application, so make sure you notice the links to the CSS and JavaScript in the head of the document. You'll be creating those two files next. Make sure you save all three files in the *same folder*.

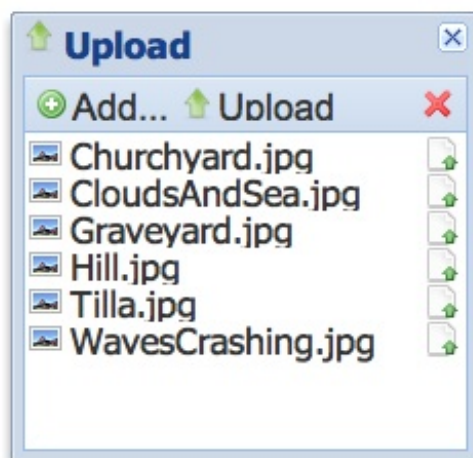
CODE TO TYPE:


```
<!doctype html>
<html lang="en">
<head>
  <title> My Photos </title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="photos.css">
  <script src="photos.js"></script>
</head>
<body>
  <nav>
    <ul>
      <li><span id="pets">Pets</span>
        <ul id="petsList">
          <li><a href="images/Tilla.jpg">Tilla</a></li>
          <li><a href="images/Pickles.jpg">Pickles</a></li>
          <li><a href="images/Jack.jpg">Jack</a></li>
        </ul>
      </li>
      <li><span id="landscape">Landscape</span>
        <ul id="landscapeList">
          <li><a href="images/Graveyard.jpg">Graveyard</a></li>
          <li><a href="images/Hill.jpg">Hill</a></li>
          <li><a href="images/Churchyard.jpg">Churchyard</a></li>
        </ul>
      </li>
      <li><span id="ocean">Ocean</span>
        <ul id="oceanList">
          <li><a href="images/WavesCrashing.jpg">Waves crashing</a></li>
          <li><a href="images/BreakingWave.jpg">Breaking wave</a></li>
          <li><a href="images/CloudsAndSea.jpg">Clouds and sea</a></li>
        </ul>
      </li>
    </ul>
  </nav>
  <div id="image">
  </div>
</body>
</html>
```

We're keeping this page very simple: we have a navigation section at the top with three nested menus, and a <div> where we'll place the image (next lesson!).

As you can see from the HTML, you're going to need images for this application. You can either use the photos from [the example](#) (to use these, right-click on the images and download them to your local machine) or use your own.

To add photos to your directory at OST, use the File Browser. Create a folder named **images** in the **/javascript 1** folder. Select the **images** folder, and then click **File**. Click **Upload File**, and upload the photos from your local machine to the **images** folder.



Save it in your `/javascript1` folder as **photos.html** and click [Preview](#)  to test the HTML. You haven't created the CSS and JavaScript yet, but you can test the links and make sure all your images are in the right places.

Add the CSS

Now it's time to add the CSS. Create a new file and add the CSS below.

Type the following into your HTML editor


```
body {
    font-family: helvetica, verdana, sans-serif;
}
ul {
    margin: 0px;
    padding: 0px;
    list-style: none;
}
ul li {
    display: inline-block;
    position: relative;
}
ul li span {
    padding: 2px 12px 2px 7px;
    background-color: #bebebe;
    display: block;
}
ul li ul {
    position: relative;
    margin: 0px;
    padding: 0px;
    left: 0px;
    top: 5px;
    width: 100%;
    visibility: hidden;
}
ul li ul li {
    padding: 2px 7px 2px 7px;
    display: block;
}
ul li ul li:hover {
    background-color: #bebebe;
}
ul li ul li a {
    display: block;
    color: black;
    text-decoration: none;
}
span:hover {
    cursor: hand;
    cursor: pointer;
    user-select: none;
}
.show {
    visibility: visible;
}
```

You don't need to understand all this CSS yet; we'll discuss the most important parts here.

OBSERVE:

```
ul li ul {  
  position: relative;  
  margin: 0px;  
  padding: 0px;  
  left: 0px;  
  top: 5px;  
  width: 100%;  
  visibility: hidden;  
}
```

ul li ul selects the nested lists; that is, the selector selects only those `` elements that are nested within `` elements that are themselves nested within `` elements (you might want to refer back to the HTML). These nested lists are the drop-down parts of the menu, that appear below each of the main headings in the menu items: Pets, Landscape, Ocean. We want the initial state of the drop-down menus to be invisible, so we are using the **visibility** CSS property to hide them until you click on the main menu item. (Try [the example](#) again to see how this works.)

Save it in your `/javascript1` folder as **photos.css** so your link in the HTML will work correctly. Switch back to **photos.html** and click **Preview** . The CSS now takes effect, so you see only the main menu items, and the drop-down menus are invisible. In addition, you'll see that the menu items are in a horizontal layout, so the drop-down menus can "drop down" below each of the main items. However, when you click on the menu items, nothing happens; that's what we need the JavaScript to do!

Before we leave the CSS, also take note of the **.show** selector:

OBSERVE:

```
.show {  
  visibility: visible;  
}
```

You know, because of the "." in front of "show", that this is a class selector. All this rule does is set the visibility property back to visible. Now we'll write some JavaScript to add this class to the appropriate drop-down menu when you click on the corresponding main menu item.

Using `setAttribute()` to Add a Class to an Element

Next, we'll use the `setAttribute()` method to update the class of an element using JavaScript:

CODE TO TYPE:

```
window.onload = init;


function init() {
    var petsSpan = document.getElementById("pets");
    var landscapeSpan = document.getElementById("landscape");
    var oceanSpan = document.getElementById("ocean");

    petsSpan.onclick = selectPets;
    landscapeSpan.onclick = selectLandscape;
    oceanSpan.onclick = selectOcean;
}

function selectPets() {
    var ul = document.getElementById("petsList");
    ul.setAttribute("class", "show");
}

function selectLandscape() {
    var ul = document.getElementById("landscapeList");
    ul.setAttribute("class", "show");
}

function selectOcean() {
    var ul = document.getElementById("oceanList");
    ul.setAttribute("class", "show");
}
```

Save it in your `/javascript1` folder as **photos.js**, switch back to **photos.html**, and click . Click on the main menu items. The drop-down menus now appear! And, because each item in the drop-down menus is a link, when you click on the image items, you'll see a new page displaying the image. We're going to change this behavior in the next lesson, so for now just use the back button to come back to the main page.

Let's walk through how the JavaScript works.

OBSERVE:

```
function selectPets() {
    var ul = document.getElementById("petsList");
    ul.setAttribute("class", "show");
}
```

If you look back at the HTML, you'll notice that each of the main menu items has an id in a `` element, and each of the drop-down menus also has an id in the `` element for that menu. So, we first add a click handler function to each of the `` elements; and when, say, the "Pets" menu item is clicked (that is, you click on the ``), the appropriate click handler is called (in this case, `selectPets()`), which gets the drop-down menu for Pets by its id, "petsList," and then uses the `setAttribute()` method to set the **class** attribute of that `` element to the **"show"** class. By setting the **class** to **"show"**, we override the **visibility** property that has the `` set to "hidden", with a new visibility property that makes it visible (remember that CSS rules that come *later* in the file override properties in *earlier* rules).

Now let's take a closer look at the `setAttribute()` method. This method is another element object method (like `appendChild()` that we used in the previous lesson), so you call it on the element whose attribute you want to set. The first argument is the name of the attribute, in this case, **"class"**, and the second argument is the value for the attribute, in this case, **"show"**. By setting the class attribute in this way, it's as if you'd typed:

```
<ul id="petsList" class="show">
```

Note that when you set the **class** attribute like this, you'll overwrite any previous value of the class attribute that element might have had. In our case, that doesn't matter, but there may be times when you want to *add*, rather than *replace*, a class, and for that, you'll need to first get the value of the attribute and then add on the new class name! (Sounds like a good exercise for the reader...).

Using `getAttribute()` to Get the Value of an Element Attribute

You might have already noticed that once you've clicked on a main menu item, the drop-down menu doesn't go away! What we'd really like is to be able to click on the main menu item a second time to make the drop-down menu disappear (this is commonly how menu items work). To implement this functionality, we need to be able to determine if a drop-down menu is visible or not when you click on the main menu item. If it is visible already, then we need to make it invisible. However, if it's invisible, then we do what we're already doing: make it visible. We can do this by checking to see if the "show" class is set on the for the drop-down menu; if it's there, we know the drop-down menu is visible, and we can make it invisible again by *removing* the "show" class from the element.

To do all that, we'll need to make use of the **getAttribute()** method. Update your "photos.js" file:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var petsSpan = document.getElementById("pets");
    var landscapeSpan = document.getElementById("landscape");
    var oceanSpan = document.getElementById("ocean");


    petsSpan.onclick = selectPets;
    landscapeSpan.onclick = selectLandscape;
    oceanSpan.onclick = selectOcean;
}

function selectPets() {
    var ul = document.getElementById("petsList");
    ul.setAttribute("class", "show");
    showHide(ul);
}

function selectLandscape() {
    var ul = document.getElementById("landscapeList");
    ul.setAttribute("class", "show");
    showHide(ul);
}

function selectOcean() {
    var ul = document.getElementById("oceanList");
    ul.setAttribute("class", "show");
    showHide(ul);
}

function showHide(el) {
    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}
```

Save it, switch back to **photos.html**, and click . Clicking once on a main menu item to display the drop-down menu, and then click again on the same main menu item—the drop-down menu should disappear. Try it with each of the main menu items.

To implement this, we consolidated all the new code into a new function, **showHide()**, since it's the same code in each case, and we pass in the correct element from each of the click handlers. So, if you click on "Pets," we'll pass in the "petsList" element.

OBSERVE:

```
function showHide(el) {  
    var ulClass = el.getAttribute("class");  
    if (ulClass == "show") {  
        // item is selected, so deselect it  
        el.setAttribute("class", "");  
    }  
    else {  
        // item is not selected, so select it  
        el.setAttribute("class", "show");  
    }  
}
```

First, we **get the current value of the class attribute using the `getAttribute()` method**. Again, this is an element object method; so we call it on the element whose attribute we want to get—if you click on **Pets**, that element is "petsList" . The first time you click on **Pets**, the "petsList" has no class at all, so the **ulClass** variable will be "". We **check to see if the ulClass variable contains the "show" string**, which it doesn't this time, so we do what we were doing before: we **set the class attribute of the "petsList" to "show."** However, if you click **Pets** again when the class attribute is already set to "show", we want to *remove* the class, so the drop-down menu will disappear. To do this, we can use **setAttribute()** again, and **set the value of the class attribute to ""**. As soon as this code runs, the "visibility" is again set to "hidden" and the drop-down menu disappears!

Use `document.querySelectorAll()` and `setAttribute()` to Set the Class of Multiple Elements

There's one more thing to do to this application before we end the lesson. Did you notice that if you click on one main menu item, like **Pets**, and then click on another, like **Landscape**, the **Pets** drop-down menu is still visible? The behavior we'd like (and the behavior we are more likely to expect) is that if you click on **Pets** and then **Landscape**, the **Pets** drop-down menu will disappear. Try [the completed example](#) again if you want to compare this behavior to the current behavior of your application.

To make this work, we need to add one more thing to the code. When you click on a main menu item like **Pets**, before we determine if we should show or hide the "Pets" menu, we need to make sure the other drop-down menus are hidden. To do this, we'll use a method to select elements from the DOM that you haven't seen before: **`document.querySelectorAll()`**. This method makes it super-easy to select elements from the DOM using CSS-like selectors.

Note

The **`document.querySelectorAll()`** method is new in HTML5 so it will only work in modern browsers. Make sure you're using the most recent version of your browser.

CODE TO TYPE:

```
window.onload = init;

function init() {
    var petsSpan = document.getElementById("pets");
    var landscapeSpan = document.getElementById("landscape");
    var oceanSpan = document.getElementById("ocean");

    petsSpan.onclick = selectPets;
    landscapeSpan.onclick = selectLandscape;
    oceanSpan.onclick = selectOcean;
}

function selectPets() {
    var ul = document.getElementById("petsList");
    showHide(ul);
}

function selectLandscape() {
    var ul = document.getElementById("landscapeList");
    showHide(ul);
}

function selectOcean() {
    var ul = document.getElementById("oceanList");
    showHide(ul);
}

function showHide(el) {
    // hide everything except what we clicked on
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] != el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}
```

Save it, switch back to **photos.html**, and click . Click **Pets**, then click **Landscape**. The Pets drop-down menu should disappear. Now click **Ocean**, and the Landscape drop-down should disappear.

Let's step through an example to see how this works. Suppose you clicked the **Pets** main menu item, so the "petsList" drop-down menu is visible. That means that the "petsList" element has a class attribute set to "show." Now, you click **Landscape**, the selectLandscape() method is called, and that method calls showHide(), and passes in the "landscapeList" element.

OBSERVE:

```
function showHide(el) {
    // hide everything except what we clicked on
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] !== el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}
```

Before adding the "show" class to the "landscapeList" element, we want to remove the "show" class from the "petsList" element. To do this, we have to find out which element has the "show" class so we can remove it. To get all of the elements that currently have the class "show," we use the `querySelectorAll()` method.

This method takes a string, which is a selector, similar to what you use in CSS to select elements for styling. We use the string `".show"` to select all the elements with the class "show." No matter what state the application is in, we should only get zero or one elements as a result, but no matter how many we get back, the result is always a collection of elements (which can be empty if *no* elements have that class). In this case, however, we know that the "petsList" element does have this class, so we'll get that element back as the result.

We **loop through all the items in the collection (which is just one)**, and we **compare each item to the one we just clicked on**. We only want to remove the "show" class from the elements that currently have it, but *not* if the element is the one we've just clicked on. Why? Because if you click on the *same* menu item (that is, instead of clicking **Landscape**, you click on **Pets** to close the drop-down menu), you'll be removing the "show" class and then the remaining code in the function will add the "show" class again, and you'll be in a state where you can never get the Pets menu to go away!

If we find a drop-down menu that is *different* from the one we've just clicked on that has the class "show," we remove the class just like we did previously, by **using `setAttribute()` to set the class attribute to ""**. So, we remove the "show" class from the "petsList" drop-down menu, which hides it. Now we're in a state where no drop-downs are displayed.

The rest of the code is the same. We get the class attribute for the drop-down menu corresponding to the menu item we clicked; in our example, that's the "landscapeList" element. This element does *not* have the "show" class (because we hadn't clicked on it previously), so we set the class attribute to "show" and it appears.

We're using the "show" class as a way to detect whether a menu item is visible or not. This is a common thing to do in web applications, and so you'll probably find yourself using `getAttribute()` and `setAttribute()` a lot to do similar things.

Try stepping through another couple of use cases to make sure you understand how the code works when the menus are in various different states.

Keep this code because you'll be using it in the next lesson!

Setting Style Directly Using JavaScript

You know how to change the style of an element using JavaScript to add or remove a class attribute from that element. Another way to manipulate the style of an element is to use the element object's **style** property. The style property is actually an object that contains some (but not all!) of the properties you can set in CSS as JavaScript properties. Because not all CSS properties are available in the style object, it's not quite as flexible as being able to set the class attribute to an arbitrary class with any CSS properties you want, but it can still come in handy!

Let's take a look at how you can set **style** directly using JavaScript, rather than by adding or removing **classes**. Neither method is better than the other, although using **classes** is often more concise, which is always a good thing.

CODE TO TYPE:

```
<!doctype html>
<html>
  <head>
    <title>Setting Style with JavaScript</title>
    <meta charset="utf-8">
    <style>
      div#container {
        display: table;
        border-spacing: 10px;
      }
      div.box {
        color: white;
        width: 200px;
        height: 200px;
        padding: 10px;
      }
      div#div1 {
        display: table-cell;
        background-color: blue;
      }
      div#div2 {
        display: table-cell;
        background-color: darkgreen;
      }
      div#div3 {
        display: table-cell;
        background-color: purple;
      }
    </style>
    <script>
      window.onload = init;

      function init() {
        var div1 = document.getElementById("div1");
        div1.onclick = changeVisibility;

        var div2 = document.getElementById("div2");
        div2.onclick = changeColor;

        var div3 = document.getElementById("div3");
        div3.onclick = changeBorder;
      }


      function changeVisibility() {
        var div1 = document.getElementById("div1");
        div1.style.visibility = "hidden";
      }
      function changeColor() {
        var div2 = document.getElementById("div2");
        div2.style.backgroundColor = "red";
        div2.style.color = "black";
      }
      function changeBorder() {
        var div3 = document.getElementById("div3");
        div3.style.borderWidth = "5px";
        div3.style.borderColor = "black";
        div3.style.borderStyle = "solid";
      }
    </script>
  </head>
  <body>
    <div id="container">

    <div class="box" id="div1">
      Click me to make me invisible!
    </div>
```

```
<div class="box" id="div2">
Click me to change my color!
</div>

<div class="box" id="div3">
Click me to change my border!
</div>

</div>
</body>
</html>
```

Save the file in the `/javascript1` folder as **style.html** and click . Try clicking on the different boxes. The first box disappears, the second box changes color, and the third box changes its border.

Let's step through how this works.

OBSERVE:

```
function init() {
  var div1 = document.getElementById("div1");
  div1.onclick = changeVisibility;

  var div2 = document.getElementById("div2");
  div2.onclick = changeColor;

  var div3 = document.getElementById("div3");
  div3.onclick = changeBorder;
}
```

First, we **set click handlers for each of the <div> elements**. Each <div> gets a different click handler, because each function will do something different.

Let's check out the **changeVisibility()** function first. When you click on "div1," the <div> disappears. We do this by setting the **visibility** CSS property using the **style** property of the **<div>** object:

OBSERVE:

```
function changeVisibility() {
  var div1 = document.getElementById("div1");
  div1.style.visibility = "hidden";
}
```

Notice that we use the string **"hidden"** to change the value of the **visibility** property. This is the same exact value you'd use for the visibility property in CSS if you were doing this using CSS.

When you click on the second <div> the color changes (both the background color and the text color).

OBSERVE:

```
function changeColor() {
  var div2 = document.getElementById("div2");
  div2.style.backgroundColor = "red";
  div2.style.color = "black";
}
```

We get the "div2" **<div>** object, and then set the **backgroundColor** and **color** properties again using the **style**. Again, notice that the **values** we set these properties to are exactly the values you'd use if you were setting them using CSS.

However, notice that the **backgroundColor** property name is a little different from what you'd use in CSS. In CSS, you set the background color of an element using the property name "background-color". But in JavaScript, background-color is not a valid variable name, right?! So, the name of the property in JavaScript is **style.backgroundColor**. This is common; for instance, font-size becomes **style.fontSize**, border-width becomes **style.borderWidth** and so

on.

OBSERVE:

```
function changeBorder() {  
    var div3 = document.getElementById("div3");  
    div3.style.borderWidth = "5px";  
    div3.style.borderColor = "black";  
    div3.style.borderStyle = "solid";  
}
```

The changeBorder() function is similar to the others. Notice that when you set the **borderWidth** property to be 5 pixels wide, you must include "px" in the **property value**, just like you would in CSS.

Experiment! Try setting the style values of various CSS properties on elements using JavaScript. It's kind of fun!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Navigating the DOM

Lesson Objectives

When you complete this course, you will be able to:

- add a click handler to multiple links.
 - add an image to your page.
 - remove elements from your page.
 - retrieve a child's parent.
 - use text nodes.
-

When you finished the previous lesson, you had a web application to show and hide a menu and display images. You can click on the images in the menu, but the link opens the image in a new page rather than in the same page where the menu is. Our goal is to open the image in the *same* page. To do that, we need to add the image to the page. We'll do that by combining some of the new things we've learned about JavaScript in the past couple of lessons: `createElement()`, `getAttribute()`, `setAttribute()`, `appendChild()`, and `querySelectorAll()`, with a method you haven't used yet, `removeChild()`, that allows you to remove elements from the DOM (the opposite of `appendChild()`).

Add a Click Handler to Multiple Links

First, make sure you have your HTML. You can use the same file from the previous lesson, or type it in again, or use the Save As icon in CodeRunner to save it with a different name. Make sure you're linking to the correct CSS and JavaScript files; if you want to create a new JavaScript file, that's fine, just remember to change the name in the `<script>` link.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> My Photos </title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="photos.css">
  <script src="photos.js"></script>
</head>
<body>
  <nav>
    <ul>
      <li><span id="pets">Pets</span>
        <ul id="petsList">
          <li><a href="images/Tilla.jpg">Tilla</a></li>
          <li><a href="images/Pickles.jpg">Pickles</a></li>
          <li><a href="images/Jack.jpg">Jack</a></li>
        </ul>
      </li>
      <li><span id="landscape">Landscape</span>
        <ul id="landscapeList">
          <li><a href="images/Graveyard.jpg">Graveyard</a></li>
          <li><a href="images/Hill.jpg">Hill</a></li>
          <li><a href="images/Churchyard.jpg">Churchyard</a></li>
        </ul>
      </li>
      <li><span id="ocean">Ocean</span>
        <ul id="oceanList">
          <li><a href="images/WavesCrashing.jpg">Waves crashing</a></li>
          <li><a href="images/BreakingWave.jpg">Breaking wave</a></li>
          <li><a href="images/CloudsAndSea.jpg">Clouds and sea</a></li>
        </ul>
      </li>
    </ul>
  </nav>
  <div id="image">
  </div>
</body>
</html>
```

The first step to getting the links to work is to add click handlers to all the links. We'll do this in a slightly more sophisticated way than how we added the click handlers to the main menu items, and it will save us typing because we can use the *same* click handler function for each link! Edit your **photos.js** file as shown:

CODE TO TYPE:

```
window.onload = init;
function init() {

    var petsSpan = document.getElementById("pets");
    var landscapeSpan = document.getElementById("landscape");
    var oceanSpan = document.getElementById("ocean");

    petsSpan.onclick = selectPets;
    landscapeSpan.onclick = selectLandscape;
    oceanSpan.onclick = selectOcean;

    var links = document.querySelectorAll("a");
    for (var i = 0; i < links.length; i++) {
        links[i].onclick = addImage;
    }
}

function selectPets() {
    var ul = document.getElementById("petsList");
    showHide(ul);
}

function selectLandscape() {
    var ul = document.getElementById("landscapeList");
    showHide(ul);
}


function selectOcean() {
    var ul = document.getElementById("oceanList");
    showHide(ul);
}

function showHide(el) {

    // deselect everything
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] != el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}

function addImage(e) {
    // we'll fill this in in the next step.
    return false;
}
```

Save it, switch to **photos.html**, and click [Preview](#) . You'll notice that now when you click on the image links, nothing happens. In fact, the `addImage()` function is being called (because we've added this function as the click handler for *all* the links), and all we're doing in `addImage()` so far is returning the value **false**. What this does is keep the browser from following the link, which would take you to a new page containing the image. We'll update `addImage()` in the next step.

For now, focus on understanding how we added the `addImage()` function as the click handler for all the links.

OBSERVE:

```
var links = document.querySelectorAll("a");
for (var i = 0; i < links.length; i++) {
  links[i].onclick = addImage;
}
```

We use the `document.querySelectorAll()` method to select all the `<a>` elements in the page. As you know from the previous lesson, this returns a **collection of elements**. In this case, it returns a collection with nine elements (because we have nine `<a>` elements in the page). So we loop through these elements and **add `addImage()` as the click handler** for each one.

Add an Image to the Page

The next step is to update our `addImage()` function to add the image to the page. We'll use methods you're already familiar with: `createElement()` and `appendChild()`. We'll also use a new object, the *event object*, to get information about which image link was clicked.

CODE TO TYPE:

```
window.onload = init;

function init() {

    var petsSpan = document.getElementById("pets");
    var landscapeSpan = document.getElementById("landscape");
    var oceanSpan = document.getElementById("ocean");

    petsSpan.onclick = selectPets;
    landscapeSpan.onclick = selectLandscape;
    oceanSpan.onclick = selectOcean;

    var links = document.querySelectorAll("a");
    for (var i = 0; i < links.length; i++) {
        links[i].onclick = addImage;
    }
}

function selectPets() {
    var ul = document.getElementById("petsList");
    showHide(ul);
}

function selectLandscape() {
    var ul = document.getElementById("landscapeList");
    showHide(ul);
}

function selectOcean() {
    var ul = document.getElementById("oceanList");
    showHide(ul);
}

function showHide(el) {
    // deselect everything
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] != el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}


function addImage(e) {
// we'll fill this in in the next step.
    var a = e.target;
    var imagePath = a.getAttribute("href");

    var image = document.createElement("img");
    image.setAttribute("src", imagePath);

    var div = document.getElementById("image");

    // add image to the div
    div.appendChild(image);
    return false;
}
```

```
}
```

Save it, switch to **photos.html**, and click **Preview** . Try clicking on some of the images. Now they should appear in the page. Each time you click, a new image is added to the page! That's not exactly what we want, but it's a good start, and we'll fix it in the next step.

OBSERVE:

```
function addImage(e) {  
  var a = e.target;  
  var imagePath = a.getAttribute("href");
```

Let's step through how this works. First, notice that our `addImage()` click handler has a parameter, `e`. This is an *event object* that contains information about the click event, including which element was clicked on. This is important for us in this version of the app. Why? Because each image link has the *same* click handler! How can we determine which link was clicked, so we know which photo to display? That's where the event object comes in. The `target` property of the event object contains the element that you clicked on to invoke the click handler function. So, we can get that element (which we know is an `<a>` element) and store it in the `a` variable.

All event handlers, like click handlers, have a parameter that gets set to the event object. What that event object contains depends on the event you're handling. If you don't need the information, you can just leave off the parameter like we did previously. But if you *do* need the event information, like we do in this example, just include a parameter in your handler definition, and the event object will be passed into it.

Once we have the correct `<a>` element, we can find out which image to show by getting the value of that element's `href` property. You know how to do this: use the `getAttribute()` method.

OBSERVE:

```
var image = document.createElement("img");  
image.setAttribute("src", imagePath);
```

Next, we create the image element using `createElement()` as usual. But for an `` element, we aren't going to see an image unless we set the `src` attribute of the image, right? So we can set the `src` attribute using `setAttribute()` to the `image path` that we got from the `href` attribute.

OBSERVE:

```
var div = document.getElementById("image");  
  
// add image to the div  
div.appendChild(image);  
return false;  
}
```

Now that we have a new `` element that points to the right image, we need to add that `` element to the DOM. We add it as a child of the `"image"` `<div>`, so we get that `<div>` using `document.getElementById()`, and use `appendChild()` to add the image.

We still need the `return false` at the end; remember, that keeps the browser from following the link to the image. We're in charge of displaying the image now, so we don't want the browser to follow the link.

Removing Elements from the Page

One last thing to add to this application: instead of appending each image you click on to the previous one, we want to see only one image at a time. That means we need to remove the previous image before adding the new image. To remove elements from the DOM, use `removeChild()`. Edit the JavaScript as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var petsSpan = document.getElementById("pets");
    var landscapeSpan = document.getElementById("landscape");
    var oceanSpan = document.getElementById("ocean");

    petsSpan.onclick = selectPets;
    landscapeSpan.onclick = selectLandscape;
    oceanSpan.onclick = selectOcean;

    var links = document.querySelectorAll("a");
    for (var i = 0; i < links.length; i++) {
        links[i].onclick = addImage;
    }
}

function selectPets() {
    var ul = document.getElementById("petsList");
    showHide(ul);
}

function selectLandscape() {
    var ul = document.getElementById("landscapeList");
    showHide(ul);
}

function selectOcean() {
    var ul = document.getElementById("oceanList");
    showHide(ul);
}

function showHide(el) {
    // deselect everything
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] != el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}

function addImage(e) {
    var a = e.target;
    var imagePath = a.getAttribute("href");


    var image = document.createElement("img");
    image.setAttribute("src", imagePath);

    var div = document.getElementById("image");
    // remove existing children
    if (div.firstChild) {
        div.removeChild(div.firstChild);
    }

    // add image to the div
}
```



```
div.appendChild(image);  
return false;  
}
```

Save it, switch to **photos.html**, and click . Try clicking on some of the image links. Now you should see just one image at a time.

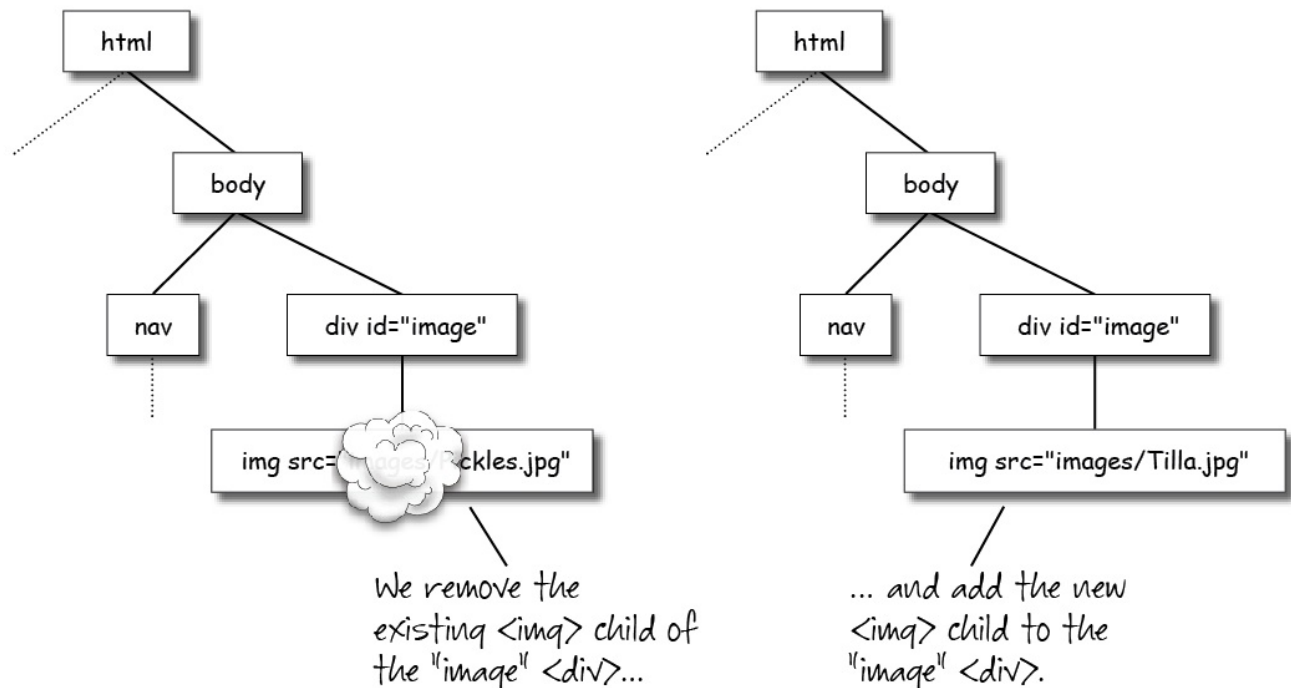
One of the challenges in removing elements from the DOM is finding the precise element you want to remove, and the parent element you need to remove it from. In this example, the `` element we want to remove doesn't have an id. However, we know that the `` element is the first (and only!) child of the "image" `<div>`, so we can find it using **`div.firstChild`**, which is the first child of the `<div>` element. We know that the `` element is the only child of the `<div>`, so that's good enough for our purposes.

OBSERVE:

```
// remove existing children  
if (div.firstChild) {  
    div.removeChild(div.firstChild);  
}
```

First we check to make sure that the **`firstChild`** exists (if it's the first time you're clicking on an image link, there will be no **`firstChild`** of the "image" `<div>` and so that will cause an error if we don't check first!). If there is a **`firstChild`**, then we can safely remove it from the `<div>` using **`removeChild()`**.

Once the old image is gone, we add the new one, and voila! The image changes in the web page from the old one to the new one.



Isn't playing with the DOM fun?

Getting a Child's Parent

You know how to find a child of an element by using the `firstChild` property, but what if you have the child and you want to find its parent? You can find the parent element of any element using the **`parentElement`** property. Let's reimplement the `addImage()` function using this property so you can see how to find a child element's parent.

CODE TO TYPE:

```
function addImage(e) {
    var a = e.target;
    var imagePath = a.getAttribute("href");

    var image = document.createElement("img");
    image.setAttribute("src", imagePath);

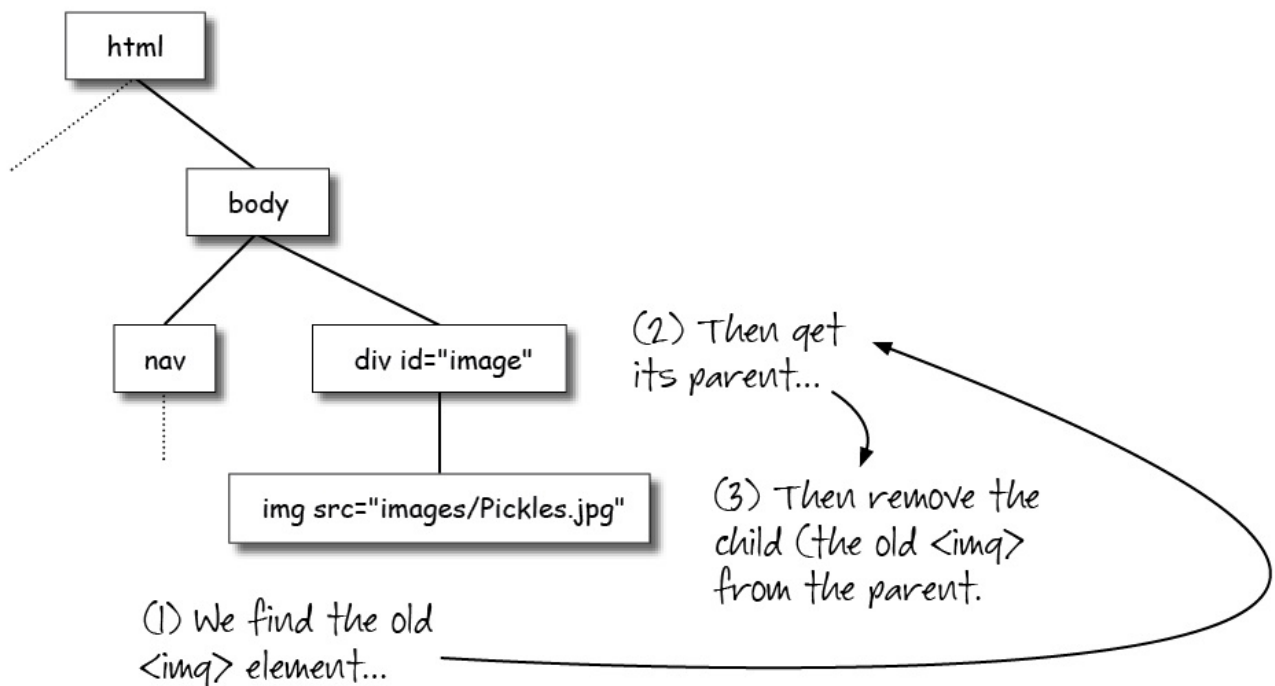
    // remove existing children
if (div.firstChild) {
    div.removeChild(div.firstChild);
}

    var oldImage = document.querySelector("img");
    if (oldImage) {
        var oldImageParent = oldImage.parentElement;
        oldImageParent.removeChild(oldImage);
    }

    // add image to the div
    var div = document.getElementById("image");
    div.appendChild(image);
    return false;
}
```

Save it, switch to **photos.html**, and click [Preview](#). You'll see exactly the same behavior as before. But now we're removing the old image by finding the ``, finding its parent element (the `<div>` element) and then removing the `` child element from the parent.

Here's how this works:



OBSERVE:

```
var oldImage = document.querySelector("img");
if (oldImage) {
    var oldImageParent = oldImage.parentElement;
    oldImageParent.removeChild(oldImage);
}
```

To find the **old image**, we use the `document.querySelector()` method, which returns only one element instead of an array of elements (because, in this example, we know that the only `` element in the page is the one we're looking for). Once we have the **existing image**, we check to make sure it really exists (`document.querySelector()` will return null if it can't find the element corresponding to the selector we pass to it), and if it does, we then find the **parent** of that element using the `parentElement` property. Once we have the parent, we can use `removeChild()` to remove the `` element we found using `document.querySelector()`. Phew! It's a bit circuitous, but it works. (In this example we have to get the "image" `<div>` anyway so we can append a new child to it, so we probably wouldn't choose to use this second method, but now you know how `parentElement` works!).

Text Nodes

You've seen how to use properties like `firstChild` and `parentElement` to access elements in the DOM tree. But you have to be a bit careful when using these properties (and other navigation properties and methods); as you may discover, the DOM tree sometimes contains unexpected children!

We kind of glossed over how text values are stored in the DOM. We showed the text values sitting in the DOM, and we've talked about setting the text content of elements with `innerHTML`, but we haven't talked about how text is actually stored in the DOM.


You might discover by accident that the DOM contains *text nodes* (objects in the DOM are often referred to as "nodes"). For instance, suppose you tried to write a program that would go through all the children of a `<div>` element and set the `backgroundColor` of each *child node* to a grey color (`#acacac`). Let's try that now.

Create a new file and add the HTML and JavaScript below:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Playing with Text Nodes </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var div = document.getElementById("article");
      for (var i = 0; i < div.childNodes.length; i++) {
        console.log(div.childNodes[i]);
        div.childNodes[i].style.backgroundColor = "#acacac";
      }
    }
  </script>
</head>
<body>
  <div id="article">
    <p id="p1">
      This is the first paragraph in my brilliant article.
    </p>
    <p id="p2">
      This is the second paragraph. Is my article getting too long?
    </p>
  </div>
</body>
</html>
```

Save it in your `/javascript 1` folder as `textnode.html` and click [Preview](#) . You see the two paragraphs in the `<div>` element, but not with a grey background!

To see why, open up the console and reload the page in the browser. You see an error message like:

"TypeError: 'undefined' is not an object (evaluating 'div.childNodes[i].style.backgroundColor = "#acacac"')"

So you're getting an error message and that's why you're not seeing the `backgroundColor`, but what on earth does that error message mean?

Let's step back and look at the code:

OBSERVE:

```
<script>
  window.onload = init;

  function init() {
    var div = document.getElementById("article");
    for (var i = 0; i < div.childNodes.length; i++) {
      console.log(div.childNodes[i]);
      div.childNodes[i].style.backgroundColor = "#acacac";
    }
  }
</script>
```

What we're attempting to do is: first, get the "article" <div> element, and then loop through all that <div> element's children, adding a grey background to each one. Notice that we use the **childNodes** property to do this loop. The **childNodes property is a collection of all the children of the <div>**. Now, if you look at the "article" <div> in the HTML, you'll note that it contains two <p> elements. So inside the for loop, we **get each child node of the <div> again using the childNodes collection, and setting its style.backgroundColor property to grey**. Okay, no problem; this should work fine, right? Except it doesn't...

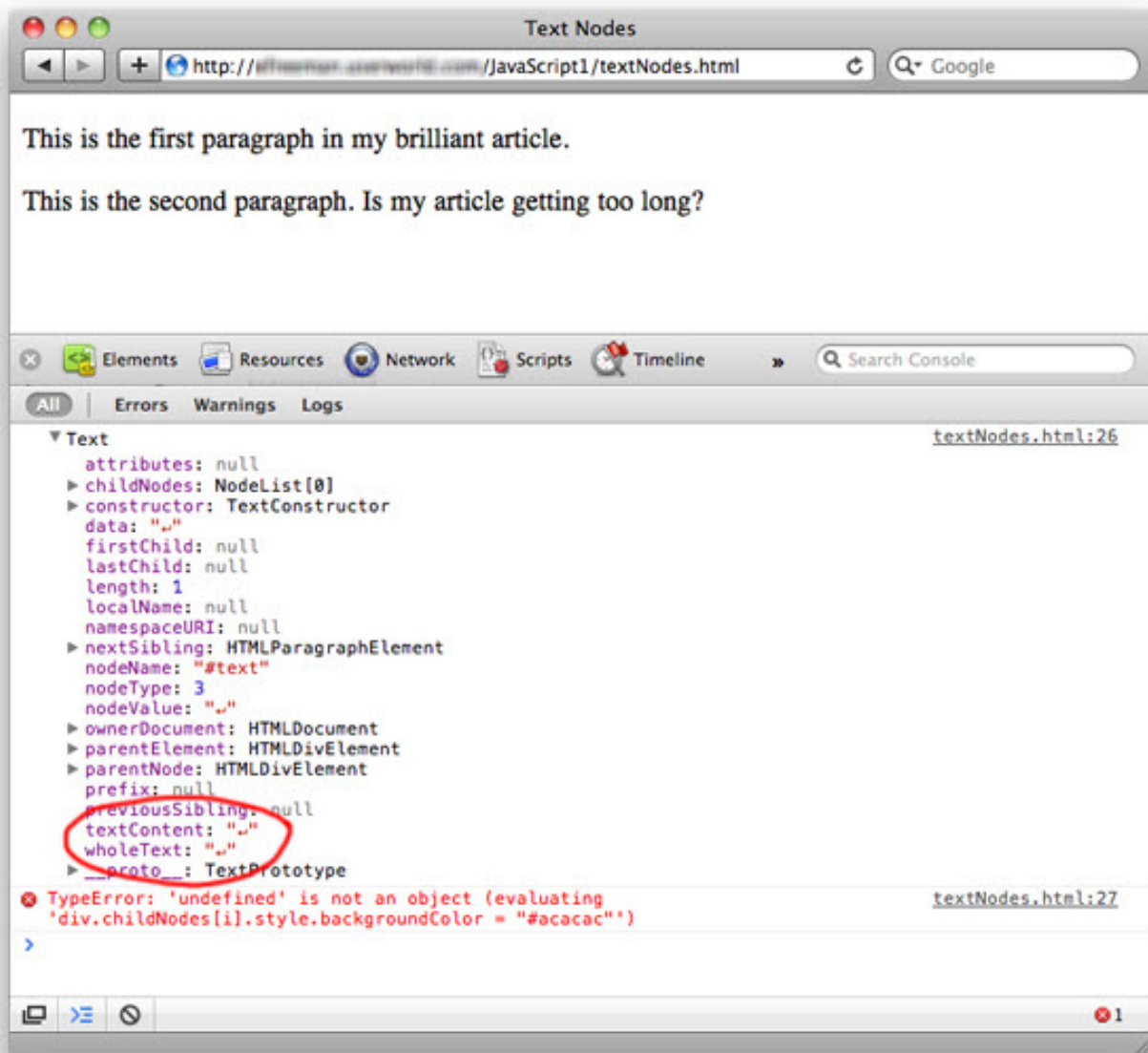
You might assume that the for loop will find two children of the <div>; the two <p> elements, right? Look at the console again. Notice that in the code, we are using console.log() to display the value of each child of the <div>, using childNode[i]. Instead of seeing the <p> element in the console, however, you will probably see **Text**.

Note

Browsers sometimes differ in how they handle text nodes, so if you aren't seeing a Text node in the console, don't worry. Just follow along to see how Text nodes work. Or try another browser to see if you get different behavior.

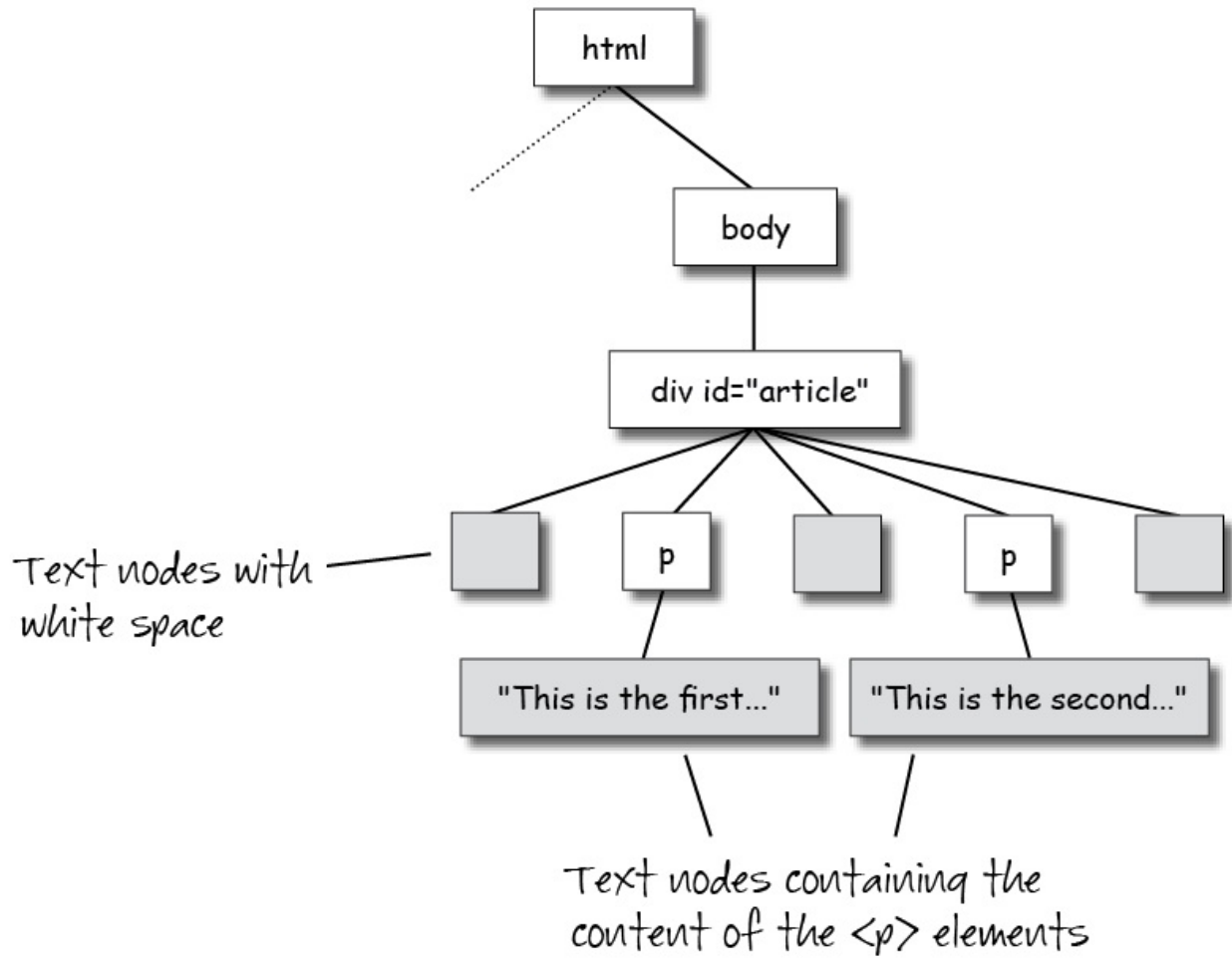
What you're seeing is what's called a *text node*. This is an object in the DOM that contains text. You might be wondering where this came from. After all, we have no text in the <div> element itself; all the text is contained in the two <p> elements that the "article" <div> element contains, right? So where is this Text node coming from?!

Again, using the console, click on the arrow next to the word "Text" to open up the Text object. Look for a property named "textContent." Do you see a little arrow symbol that looks like a return key? Here's what it looks like in Safari:



Turns out that the DOM stores white space as well as actual text content as text nodes. These text nodes are objects in the DOM just like element objects, and just like text nodes that contain actual content! That's no problem, unless you're expecting to get an element node and you get a text node by mistake and encounter an error like this one. And you're getting the error because you can't set the background color of a text node (or any other style property for that matter); *you can only set style properties on element objects!* In fact, text nodes are quite different from element objects (or "element nodes"), so you have to be sure you test for them—avoid them altogether (which you'll see how to do in a moment).

Here are the text nodes in the DOM tree for this example:




You can access the content of the `<p>` elements using text nodes; to do that, just change your JavaScript to access the `childNodes` of one of the `<p>` elements instead of the `<div>`:

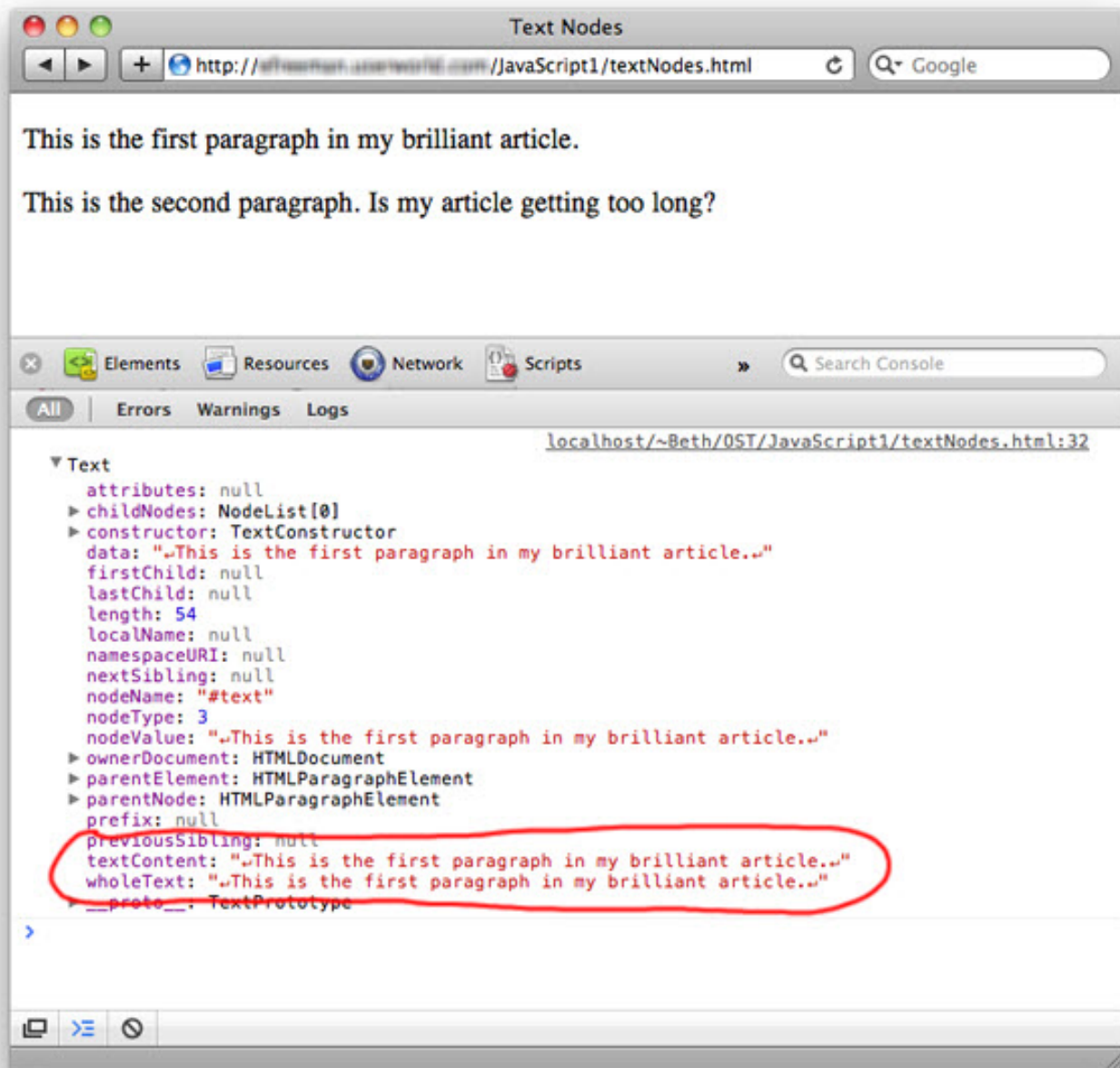
CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Playing with Text Nodes </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var div = document.getElementById("article");
      for (var i = 0; i < div.childNodes.length; i++) {
        console.log(div.childNodes[i]);
        div.childNodes[i].style.backgroundColor = "#acacac";
      }

      var p = document.getElementById("p1");
      for (var i = 0; i < p.childNodes.length; i++) {
        console.log(p.childNodes[i]);
      }
    }
  </script>
</head>
<body>
  <div id="article">
    <p id="p1">
      This is the first paragraph in my brilliant article.
    </p>
    <p id="p2">
      This is the second paragraph. Is my article getting too long?
    </p>
  </div>
</body>
</html>
```

Save it and click [Preview](#) . Now, the Text object you see in the console contains real text content. Take a look by again opening up the Text object in the console, and looking for that **textContent** property. Now you should see something more like this:



Check that the value of the **textContent** property you're seeing in the console matches what you wrote as the content of the first `<p>` element in the HTML.


Avoiding Text Nodes Using `getElementById()` or `querySelectorAll()`

So what's a solution to our original problem? In other words, if we want to set the `backgroundColor` of all the children of the "article" `<div>` to grey, how do we do it without getting hung up by these pesky text nodes? Easy! You already know a couple of different ways of doing this using `document.getElementById()` and `document.querySelectorAll()`, right? Let's see how you might do this using `document.querySelectorAll()`.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Playing with Text Nodes </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var paras = document.querySelectorAll("div#article p");
      for (var i = 0; i < paras.length; i++) {
        paras[i].style.backgroundColor = "#acacac";
      }
    }
  </script>
</head>
<body>
  <div id="article">
    <p id="p1">
      This is the first paragraph in my brilliant article.
    </p>
    <p id="p2">
      This is the second paragraph. Is my article getting too long?
    </p>
  </div>
</body>
</html>
```

Save it and click . Now when you load the page, your paragraphs should have a grey background color, just like you would expect.

We used **document.querySelectorAll("div#article p")** to get the `<p>` elements from the DOM. This ensures that we don't get any text nodes; we only get element objects in the resulting collection, so we can safely set the background color to grey without getting an error message. The selector "div#article p" selects all the `<p>` elements that are children of the `<div>` element with the id "article."

Query selectors are a powerful way of selecting DOM objects from your page to manipulate. You'll probably find you use this method a lot!

Note

Make sure you test your code in a variety of browsers. At this point, the only browser versions you still have to worry about when using the query selector methods are IE 6 and IE 7. IE 8+ and all the modern versions of the other major browsers support these methods. If you want to support these older browsers, you'll need to use **document.getElementById()** or **document.getElementsByTagName()** instead.

Now you know the basics of how to navigate the DOM and manipulate your web pages with JavaScript. With these tools you can create pretty amazing web applications!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Final Project--The Amazing Box Generator!

Lesson Objectives

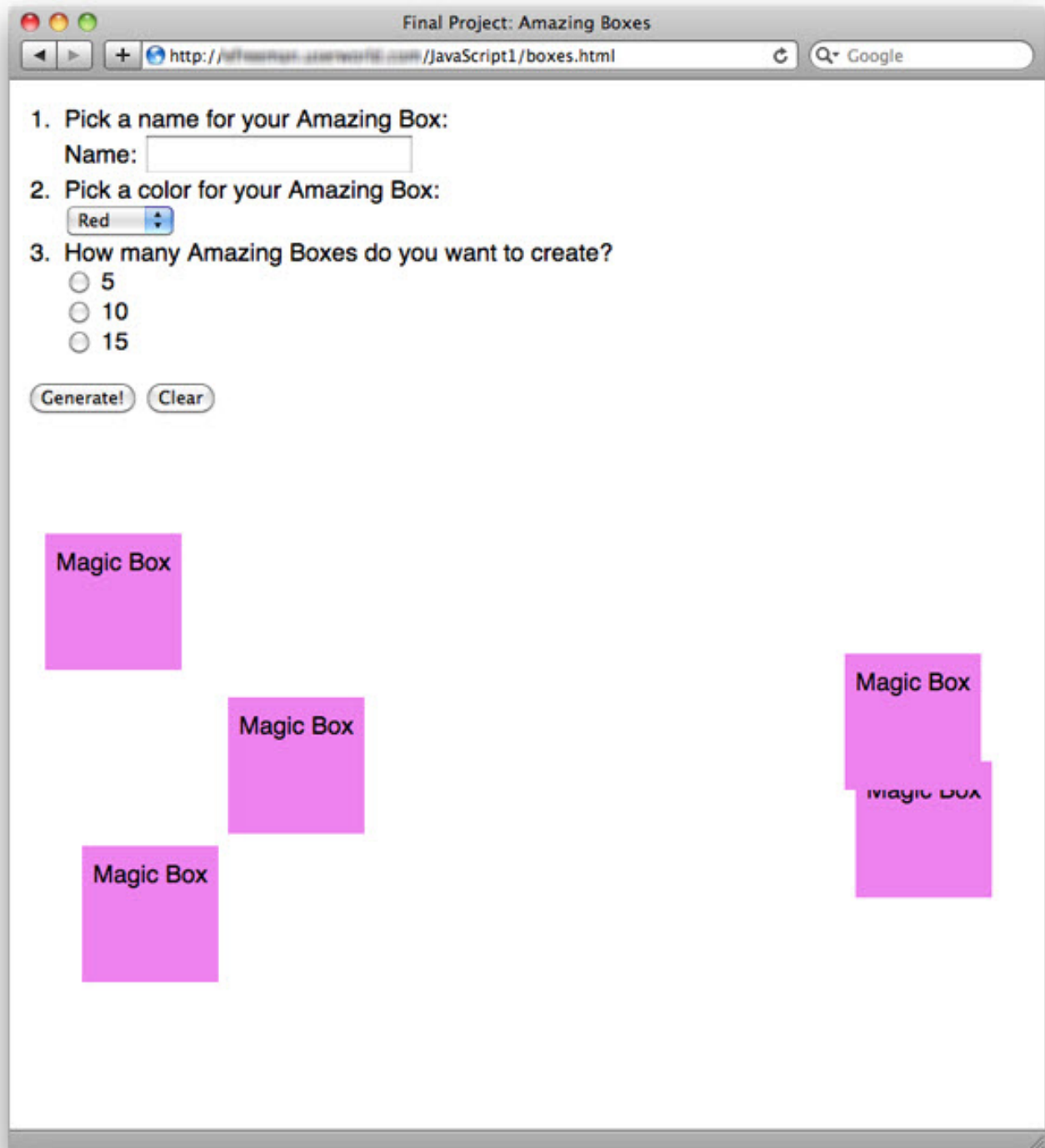
When you complete this course, you will be able to:

- create a web application called "Amazing Boxes."
- get users to select color, number of boxes to generate, and name the boxes.
- create a menu of colors in a form.
- create a box object with properties.
- add divs that represent boxes to DOM.
- style the divs.
- generate a button.
- clear the button and remove all divs.
-

You've come a long way in this course, from not knowing any JavaScript, to knowing quite a lot! Congrats. This last lesson is about the final project, which is to create a web application called "Amazing Boxes." What are amazing boxes, you might ask? Read on to find out...

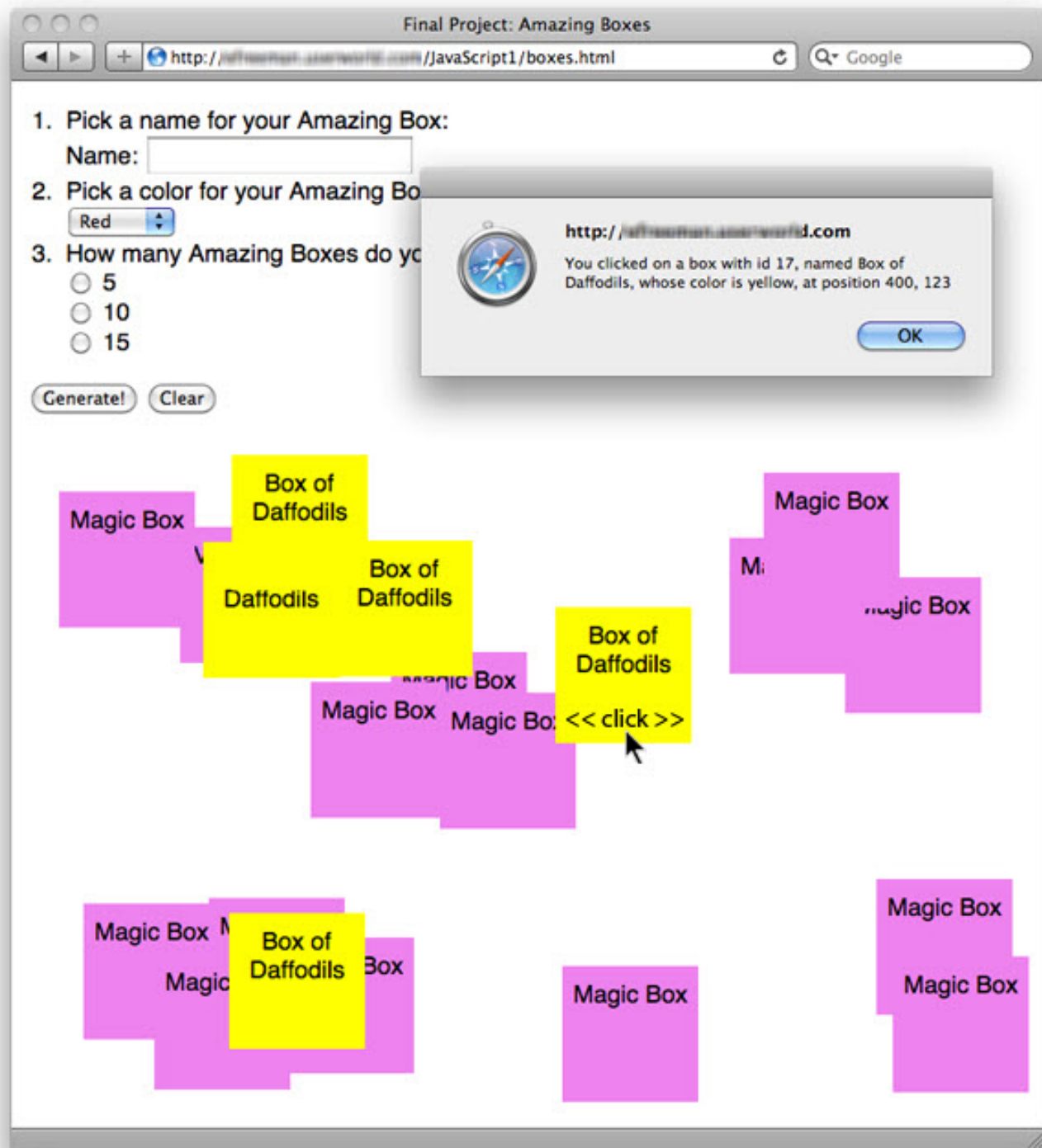
Amazing Boxes

This app has a web page with a form that prompts you for three things: the name for your amazing boxes, a color for your boxes, and how many you want to generate. Let's say you enter "Magic Box" for the name, you choose "Violet" for the color, and you select 5 as the number of boxes to create. Here's what you'll see when you click the **Generate!** button:



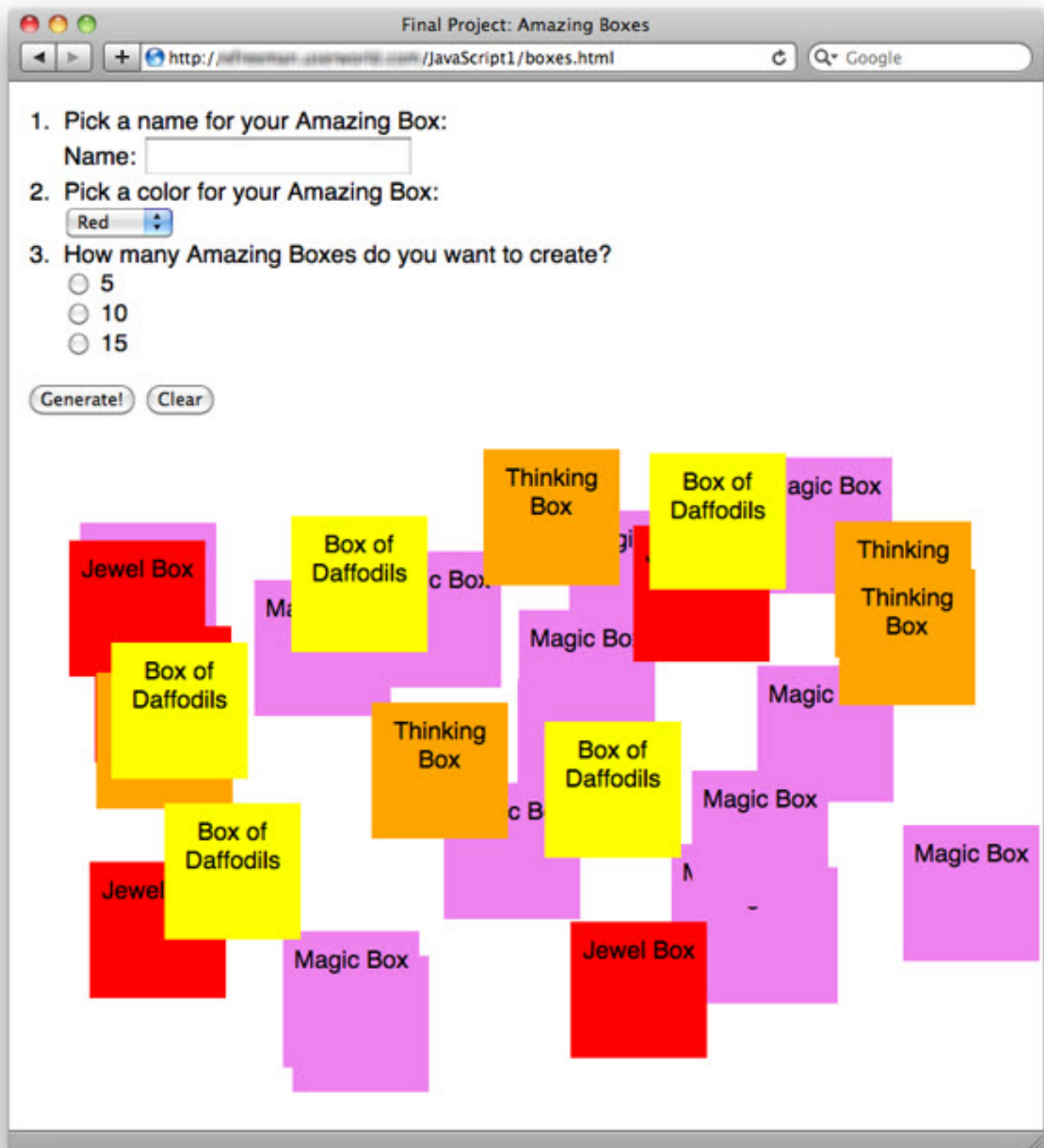
Five boxes (implemented as `<div>` elements) are added to the page, with "violet" as their background color, the name of the box as the content of the box, and their positions randomly selected inside the part of the page where they have been added.

You can go ahead and generate more amazing boxes (no need to reload the page); you could then enter "Box of Daffodils" for the name, choose "Yellow" for the color, and choose 5 again for the number. When you click **Generate!** you'll see:



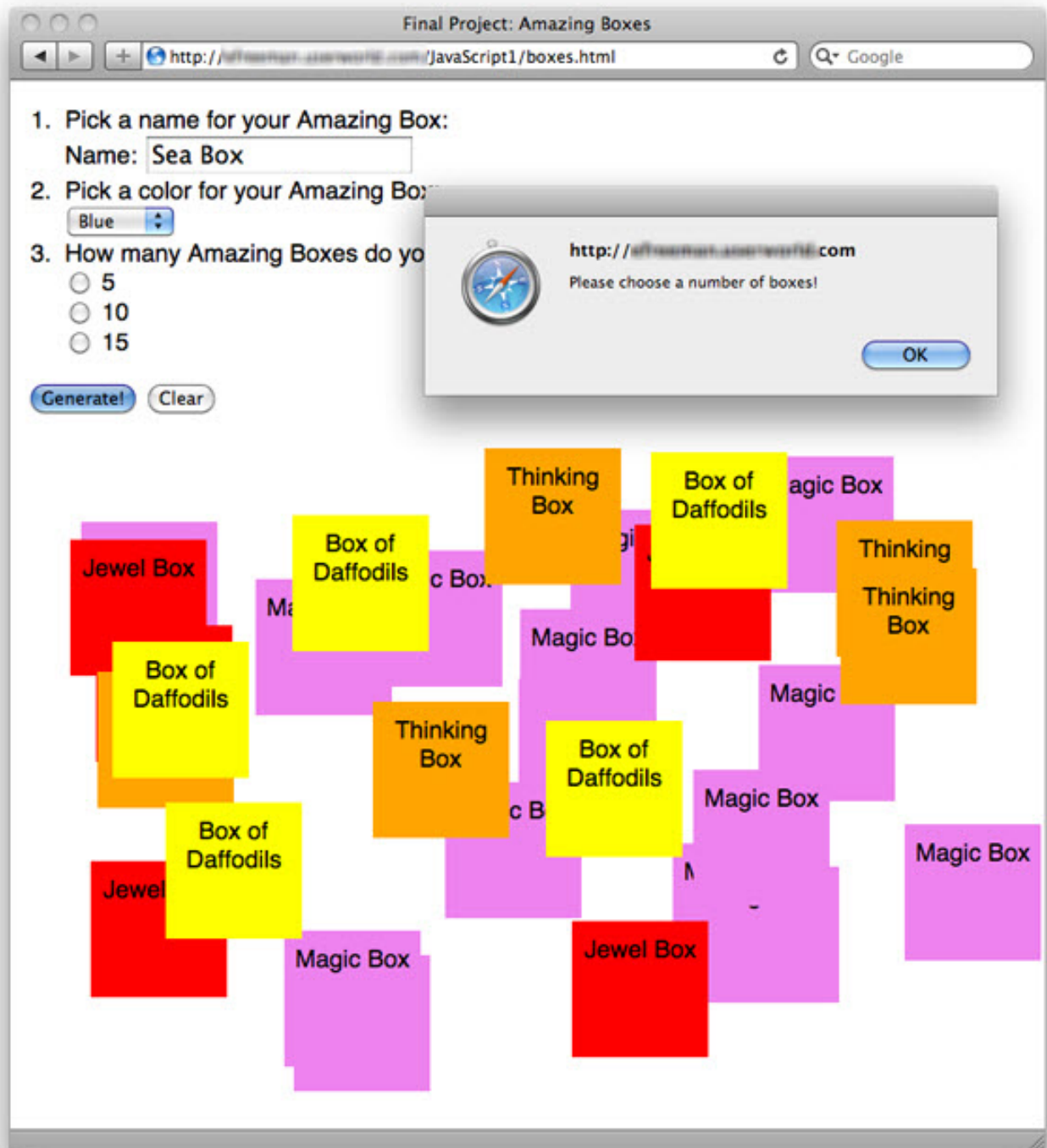
Notice that the new boxes are *added* to the page and that the "Magic Boxes" are still there. Also notice that if you click on one of the boxes, you see information about the box: its id (a unique number), name, color, and position in x, y (left, top) pixels, in the page.

You can keep adding more boxes, as many as you like:



A nice feature to add to the app is to reset the form after clicking the **Generate!** button (like we did in an earlier lesson) so it's easier for the user to enter the next set of values.

If you forget to enter a name or choose a number, you will see an alert indicating you need to enter the value:



Finally, if you click the **Clear** button, all the boxes will be erased! You can start over and add new boxes at that point if you want.

Final Project: Amazing Boxes

http://www.dreamhost.com/JavaScript1/boxes.html

Google

1. Pick a name for your Amazing Box:
Name:
2. Pick a color for your Amazing Box:
3. How many Amazing Boxes do you want to create?
☐ 5
☐ 10
☐ 15

We'll give you the HTML and CSS for this project (aren't we nice?!), and it's your job to write the JavaScript. You already have just about everything that you need; we'll fill in a couple of pieces that will help after you take a look at the HTML and the CSS.

The HTML for Amazing Boxes

Here is the HTML for the application:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Final Project: Amazing Boxes </title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="boxes.css">
  <script src="boxes.js"></script>
</head>
<body>
  <form id="data">
    <ol>
      <li>Pick a name for your Amazing Box: <br>
        <label for="name">Name: </label>
        <input type="text" id="name" size="20">
      </li>
      <li>Pick a color for your Amazing Box: <br>
        <select id="color">
          <option value="red">Red</option>
          <option value="orange">Orange</option>
          <option value="yellow">Yellow</option>
          <option value="green">Green</option>
          <option value="blue">Blue</option>
          <option value="indigo">Indigo</option>
          <option value="violet">Violet</option>
        </select>
      </li>
      <li>How many Amazing Boxes do you want to create?<br>
        <input type="radio" id="five" name="amount" value="5">
        <label for="five">5</label><br>
        <input type="radio" id="ten" name="amount" value="10">
        <label for="ten">10</label><br>
        <input type="radio" id="fifteen" name="amount" value="15">
        <label for="fifteen">15</label><br>
      </li>
    </ol>
    <p>
      <input type="button" id="generateButton" value="Generate!">
      <input type="button" id="clearButton" value="Clear">
    </p>
  </form>
  <div id="scene">
  </div>
</body>
</html>
```

Save this with an appropriate name (we suggest **boxes.html**) in your javascript homework folder. We provide links to the **boxes.css** and **boxes.js** files in the <head>; use these names for the files, or change them here to the names you use.

We have a form with three different input types: text input, select, and radio button input. You'll need to make sure that the text input and radio button inputs are valid (the color will always default to the first item in the list, so you don't technically need to check that one). Also, we have two buttons: one to generate the boxes, and one to clear the scene. And finally, we have a "scene" <div> element at the very bottom; this is where all the boxes are going to go!

Look carefully over the HTML until you're familiar with each part.

The CSS for Amazing Boxes

Here is the CSS for the app:

CODE TO TYPE:

```
html, body {
  width: 100%;
  height: 90%;
  margin: 0px;
  padding: 0px;
  font-family: Helvetica, Arial, sans-serif;
}

input#generateButton {
  margin-left: 15px;
}

div#status {
  margin-left: 15px;
}

div#scene {
  position: relative;
  width: 100%;
  height: 80%;
  margin: 0px;
  padding: 0px;
}

div.box {
  position: absolute;
  width: 100px;
  height: 90px;
  padding-top: 10px;
  text-align: center;
  overflow: hidden;
}
```

Save it in your javascript homework folder as **boxes.css**, or again, use any name you want and change the link in the HTML file to the appropriate name. The most important two rules in the CSS are the **div#scene** rule and the **div.box** rule. Note that the "scene" <div> is positioned relative, which means we can position the box <div>s inside the "scene" <div> using absolute positioning, and they will be positioned relative to the top left corner of the "scene" <div>. That way, the boxes don't sit on top of the form!

When you add the <div> elements that are going to represent the boxes, you'll need to add the class "box" to each <div> element that represents a box. You already know how to do this.

The <div> elements with the class "box" will be absolutely positioned, which means they can be positioned using the **top** and **left** properties. You can make these values anything between 0 and the height/width of the "scene" <div> (which we'll show you how to do a moment).

Notice that the boxes are 100px by 100px (with 10px of padding on the height above the name of the box). The text is aligned to the center and any overflow text is hidden. That means when you add the name of the box as the content of the <div>, it will appear in the center of the box.

Positioning an Absolutely Positioned Element

You will position each box (that is, each <div> element with the class "box") within the "scene" <div> by randomly creating an x, y (left, top) position for the box. Here's how you can do that:

OBSERVE:

```
var sceneDiv = document.getElementById("scene");
var x = Math.floor(Math.random() * (sceneDiv.offsetWidth-101));
var y = Math.floor(Math.random() * (sceneDiv.offsetHeight-101));
```

We first get the **"scene"** <div> from the DOM, and then use its computed width and height to generate a random number x, between 0 and the **width of the <div>**, and a random number y, between 0 and the **height of the <div>**. The computed width and height of the <div> are figured out by the browser once the page is rendered, and you can get

the values using the <div>'s **offsetWidth** and **offsetHeight** properties. We don't want the boxes sitting halfway off the page at the side or bottom, so we reduce the possible starting point by the width + 1 of the box <div>. Thus, if the computed width of the "scene" <div> is 600px, the maximum x starting point for the left point of the box is 499 so the box will fit into the remaining part of the screen.

Once you have an x and y position, how do you update the style to set the position for the <div>? You can use the **style** property, which you already know how to do. However, there's one little tricky thing with the **top** and **left** properties you should know about. Here's what you'd do in CSS if you wanted to set these properties to 10, 10:

OBSERVE:

```
div.box {  
  left: 10px;  
  top: 10px;  
}
```

So, in JavaScript, to set those same values (given x and y as computed above), you'd write:

OBSERVE:

```
div.style.left = x + "px";  
div.style.top = y + "px";
```

We have to add the "px" after the values so the style properties for left and top match what you'd write in the CSS.

The Amazing Boxes Application Requirements

That's everything you need to build the app. Feel free to add any additional functions and variables you need. **Good luck!** Open the Objective to submit your files.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*